Инструментарий для вероятностной верификации на основе многокорневых диаграмм решений*

Д. Ю. Бугайченко
DmitryBugaychenko@math.spbu.ru

Кафедра информатики,
Санкт-Петербургский
государственный университет

В данной работе мы предлагаем инструментарий для проведения вероятностной верификации, основанный на многокорневых бинарных диаграммах решений. Инструмент реализован в виде платформо-независимой C++ библиотеки, предоставляющей объектно-ориентированный интерфейс для формирования моделей и верификации свойств. В качестве основы для реализации операций с многокорневыми бинарными диаграммами решений используется библиотека BddFunctions. В работе приводятся результаты экспериментального сравнения предложенного инструмента с популярным инструментом вероятностной верификации PRISM.

Ключевые слова: вероятностная верификация, бинарные диаграммы решений, вероятностная логика ветвящегося времени.

* Работа выполнена при поддержке РФФИ (грант № 09-01-00525-а).

Введение

Область применения различных информационных технологий чрезвычайно широка и продолжает постоянно расширяться. При этом постоянно растет сложность информационных систем, а также цена ошибок в них, что заставляет как научное, так и индустриальное сообщества уделять все больше внимания качеству разрабатываемых систем. Основным инструментом контроля качества является тестирование с помощью «вручную» подготовленных тестовых сценариев, что является достаточно затратным процессом и не может гарантировать корректность работы системы во всех случаях. Автоматизация выполнения тестовых сценариев позволяет несколько удешевить процесс, но сама по себе требует дополнительных начальных затрат.

Идея более полной автоматизации процесса проверки корректности поведения системы возникла достаточно давно и реализована в двух основных направлениях: верификация (model checking) [3] и генерация тестовых сценариев на основе формальной спецификации системы [9] или анализа кода системы [18]. Генерация тестовых сценариев активно применяется для проверки корректности отдельных модулей программных систем [13], в то время как верификация чаще используется для проверки корректности дизайна аппаратных систем [20] и протоколов [5, 15] (хотя есть примеры и успешной верификации программных систем [6]). При этом оба подхода часто основываются на одних и тех же структурах данных — *бинарных диаграммах решений* [7], позволяющих эффективным образом представить множество состояний системы, проследить все возможные пути выполнения и идентифицировать потенциальные проблемы.

На практике наибольшее распространение получила *строгая* верификация, однозначно отвечающая на вопрос, выполнено ли некоторое свойство для системы или нет. Однако существует насущная проблема и в решении задачи *вероятностной* верификации, способной оценить вероятность того, что для системы выполнено некоторое свойство. При рассмотрении вероятностных моделей бинарные диаграммы решений неприменимы, так как позволяют представлять только булевы функции и четкие множества. Для решения этой проблемы используются *многотерминальные* бинарные диа-

граммы решений [12], расширяющие бинарные диаграммы решений дополнительными терминальными вершинами. Однако такая модификация диаграмм решений ведет к резкому увеличению вариативности структуры диаграммы [8], что не позволяет эффективно использовать повторяющиеся блоки и ведет к росту потребления памяти. В результате, для вероятностной верификации разработчикам приходится искать альтернативные алгоритмы (например, статистическое моделирование [19]) и подходящие структуры данных (например, разреженные матрицы [10]). В работе [14] приведено сравнение существующих инструментов вероятностной верификации, показывающее, что алгоритмы, основанные на разреженных матрицах, во многих случаях превосходят алгоритмы, основанные на диаграммах решений, тогда как статистические алгоритмы работают нестабильно и могут дать неверный ответ в некоторых случаях.

В работе [8] предложена альтернативная модификация диаграмм решений — *многокорневые* бинарные диаграммы решений. В этом случае для расширения множества значений используются несколько бинарных диаграмм решений и соответствующая двоичная кодировка. Полученные структуры позволяют использовать память более эффективно по сравнению с многотерминальными диаграммами. Однако если в случае многотерминальных диаграмм все операции целевого множества непосредственно переводятся в операции с отдельными терминалами, то в случае многокорневых диаграмм необходимо строить соответствие с учетом выбранной кодировки. В работе [1] описаны операции с целочисленными функциями, а также приведены экспериментальные результаты, показывающие преимущества многокорневых диаграмм по сравнению с многотерминальными. Эти результаты расширены в работе [2], где предложены алгоритмы для операций с дискретными случайными величинами, представленными в виде многокорневых диаграмм решений.

В данной работе мы предлагаем инструментарий для вероятностной верификации, использующий многокорневые бинарные диаграммы решений в качестве структур данных. В *разделе 1* работы даны основные понятия, необходимые для её понимания, описаны некоторые способы спецификации вероятностных моделей и свойств, а также алгоритмы их проверки. *Раздел 2* содержит опи-

сание предлагаемого инструментария, его архитектуры и способов использования. В *разделе 3* приведено сравнение производительности и результатов предлагаемого инструментария с популярным инструментом PRISM [16] на двух примерах: синхронном выборе лидера в кольце процессоров [11] и упрощенном варианте процесса рождения и смерти [17].

1. Основные понятия

Для проведения процесса верификации необходимо определить три основные сущности: *модель*, описывающую систему, для которой проводится верификация, *спецификацию* в виде набора свойств, которым должна удовлетворять модель (а следовательно, и исходная система), и *алгоритм верификации*, проверяющий, выполнены ли эти свойства для данной модели.

При проведении строгой верификации в качестве модели, как правило, используют вариацию модели Крипке, представляющую собой систему в виде отношения $S \times S$, где S есть множество состояний системы. Если некоторая пара состояний (s, s') входит в это отношение, то это означает, что система может перейти из состояния s в состояние s'. Для спецификации свойств используют различные варианты темпоральных логик, расширяющих классическую логику операторами «в следующий момент», «когда-нибудь» и «всегда», а также кванторами «на любом пути» и «существует путь». При этом для проверки свойств используются алгоритмы поиска неподвижной точки: по данному множеству состояний строится множество возможных результатов, объединяется с исходным и повторяется до тех пор, пока множество состояний не стабилизируется.

Для вероятностной верификации, по очевидным причинам, нужны другие инструменты. В качестве модели системы, как правило, используются Марковские цепи или процессы, для спецификации свойств используют темпоральную логику с квантором «с вероятностью не более(не менее) чем ...», а задачу верификации сводят к задаче решения системы линейных уравнений и используют один из итеративных методов для её решения.

В данной работе в качестве способа моделирования вероятностных систем выбрана дискретная цепь Маркова с дискретным временем, представляющаяся парой $< S, \pi >$, где S есть конечное непу-

стое множество состояний системы, а $\pi : S \times S \to [0,1]$ есть распределение вероятностей изменения состояний системы (матрица вероятностей перехода). При этом для π выполнено следующее условие:

$$\forall s \in S : \sum_{s' \in S} \pi(s, s') = 1,$$

что позволяет называть эту матрицу *стохастической*.

Моделируемая дискретной цепью Маркова система совершает переходы в определенные моменты времени $t_1, t_2, \ldots$, проходя через последовательность состояний $s_1, s_2, \ldots$ В каждый момент времени t_i вероятность перехода в следующее состояние s_{i+1} зависит исключительно от текущего состояний s_i и определяется как $\pi(s_i, s_{i+1})$.

Состояние системы, как правило, формируется из набора параметров разного типа: $\{x_i \in X_i\}_{i=1}^n$. В этом случае множество S является декартовым произведением типов всех параметров:

$$S = X_1 \times \ldots \times X_n.$$

Для спецификации свойств дискретных цепей Маркова используется логика Probabilistic computational tree logic (PCTL). Синтаксис этой логики определяется следующими правилами:

$$\begin{aligned} \psi &::= \bigcirc \phi \,|\, \phi \,\mathcal{U}\, \phi \,|\, \phi \mathcal{U}^k \phi; \\ \phi &::= \chi \,|\, \neg\, \phi \,|\, \phi \,\wedge\, \phi \,|\, P^{[<=,>=]\,p} \psi. \end{aligned} \tag{1}$$

Формулы ϕ называются *формулами состояний* и могут быть проверены относительно отдельного состояния системы s. Формулы состояний строятся из *атомарных формул* χ, определяемых в зависимости от предметной области (например, при работе с численными параметрами можно использовать стандартные предикаты сравнения и арифметические функции $x_1 > x_2 + 4$) и объединенных с помощью стандартных логических связок $\neq$ и $\wedge$.

Формулы ψ называются *формулами пути* и могут быть проверены относительно последовательности состояний системы $s_1, s_2, \ldots$ Для определения формул пути используются операторы «в следующий момент ϕ» $\bigcirc\phi$, «когда-нибудь ϕ_2, а до этого ϕ_1» $\phi_1 \mathcal{U} \phi_2$ и «не более чем через k шагов ϕ_2, а до этого ϕ_1» $\phi_1 \mathcal{U}^k \phi_2$.

Для стыковки формул путей и состояний используется квантор «с вероятностью не более (не менее) p выполнено ψ» $P^{[<=,>=]\,p}\psi$,

интерпретируемый как истинный для тех состояний, где вероятность выполнения заданной формулы пути ψ удовлетворяет условию. Для того чтобы определить эту вероятность, введем функцию $prob^n : S^n \to [0,1]$, вычисляющую для пути длины n его вероятность как

$$prob^n((s_1,\ldots,s_n)) = \prod_{i=1}^{n-1} \pi(s_i, s_{i+1}),$$

а также множество всех путей, начинающихся в состоянии s:

$$Path^n(s)\{(s_1,\ldots,s_n) \in S^n \mid s_1 = s\}.$$

Формально семантику PCTL относительно цепи Маркова $< S, \pi >$ можно определить следующим образом:

1) $< S, \pi >, s \models \neg\phi$ тогда и только тогда, когда не $< S, \pi >, s \models \phi$;
2) $< S, \pi >, s \models \phi_1 \wedge \phi_2$ тогда и только тогда, когда $< S, \pi >, s \models \phi_1$ и $< S, \pi >, s \models \phi_2$;
3) $< S, \pi >, (s_1,\ldots,s_n) \models \bigcirc\phi$ тогда и только тогда, когда $n >= 2$ и $< S, \pi >, s_2 \models \phi$;
4) $< S, \pi >, (s_1,\ldots,s_n) \models \phi_1 \mathcal{U} \phi_2$ тогда и только тогда, когда существует $j \in \{1,\ldots,n\}$, такое, что $< S, \pi >, s_j \models \phi_2$, и для любого $i < j$ выполнено $< S, \pi >, s_i \models \phi_1$;
5) $< S, \pi >, (s_1, s_2, \ldots) \models \phi_1 \mathcal{U}^k \phi_2$ тогда и только тогда, когда существует $j \in \{1,\ldots,max(k+1,n)\}$, такое, что $< S, \pi >, s_j \models \phi_2$, и для любого $i < j$ выполнено $< S, \pi >, s_i \models \phi_1$;
6) $< S, \pi >, s \models P^{[<=,>=]\,p}\psi$ тогда и только тогда, когда

$$\lim_{n\to\infty} \left(\sum_{\psi^n(s)} prob^n\left(((s_1,\ldots,s_n)\right) \right) [<=,>=]\, p,$$

где $\psi^n(s) = \{(s_1,\ldots,s_n) \in Path^n(s) \mid < S, \pi >, (s_1,\ldots,s_n) \models \psi\}$ есть множество путей длины n, начинающихся в s и удовлетворяющих ψ.

При проверке некоторого свойства ϕ относительно модели $< S,\ \pi >$ наибольшую сложность представляет последнее правило, переводящее формулу пути в формулу состояния, а именно вычисление предела. Способы его вычисления отличаются в зависимости от того, какая формула пути использована внутри.

При интерпретации оператора $\bigcirc\phi$ значение имеют только пути длины 2, так как учитывается только следующее за текущим состояние. В результате нужное значение вычисляется по формуле

$$p_s(\bigcirc\phi) = \sum_{s' \in \phi} \pi(s, s')$$

как сумма вероятностей перехода из состояния s во все состояния s', удовлетворяющие ϕ.

При использовании оператора $\phi_1 \mathcal{U}^k \phi_2$ аналогично, значение имеют только пути длины $k+1$, и вероятность определяют по рекурсивной формуле:

$$p_s(\phi_1 \mathcal{U}^k \phi_2) = \begin{cases} 1, & < S, \pi >, s \models \phi_2 \\ \sum\limits_{s' \in S} \pi(s, s') \cdot p_{s'}(\phi_1 \mathcal{U}^{k-1} \phi_2), & k > 1, < S, \pi >, s \models \phi_1, \\ 0, & otherwise \end{cases}$$

где $p_s(\phi_1 \mathcal{U}^k \phi_2)$ есть искомое значение.

Аналогичное рекурсивное соотношение должно выполняться и для неограниченного оператора $\phi_1 \mathcal{U}^k \phi_2$, что позволяет свести вычисление $p_s(\phi_1 \mathcal{U} \phi_2)$ к решению системы линейных уравнений относительно переменных $\{x_s\}_{s \in S}$ вида:

$$x_s = \begin{cases} 1, & < S, \pi >, s \models \phi_2 \\ \sum\limits_{s' \in S} \pi(s, s') \cdot x_{s'}, & < S, \pi >, s \models \phi_1. \\ 0, & otherwise \end{cases}$$

Полученное в результате решения этой системы значение x_s и будет являться значением $p_s(\phi_1 \mathcal{U} \phi_2)$.

На практике вычисление описанных выше значений «в лоб», как правило, невозможно, так как количество операций в таком случае будет пропорционально количеству состояний системы $|S|$ (что для системы с двумя 32-битными целочисленными параметрами уже равно 2^{64}). Вместо этого вычисление сводят к умножению матрицы вероятностей переходов (иногда несколько измененной) и вектору вероятностей для каждого состояния. Будучи представленными в виде диаграмм решений, эти матрица и вектор во многих случаях позволяют вычислить результат умножения за приемлемое время.

2. Инструментарий вероятностной верификации

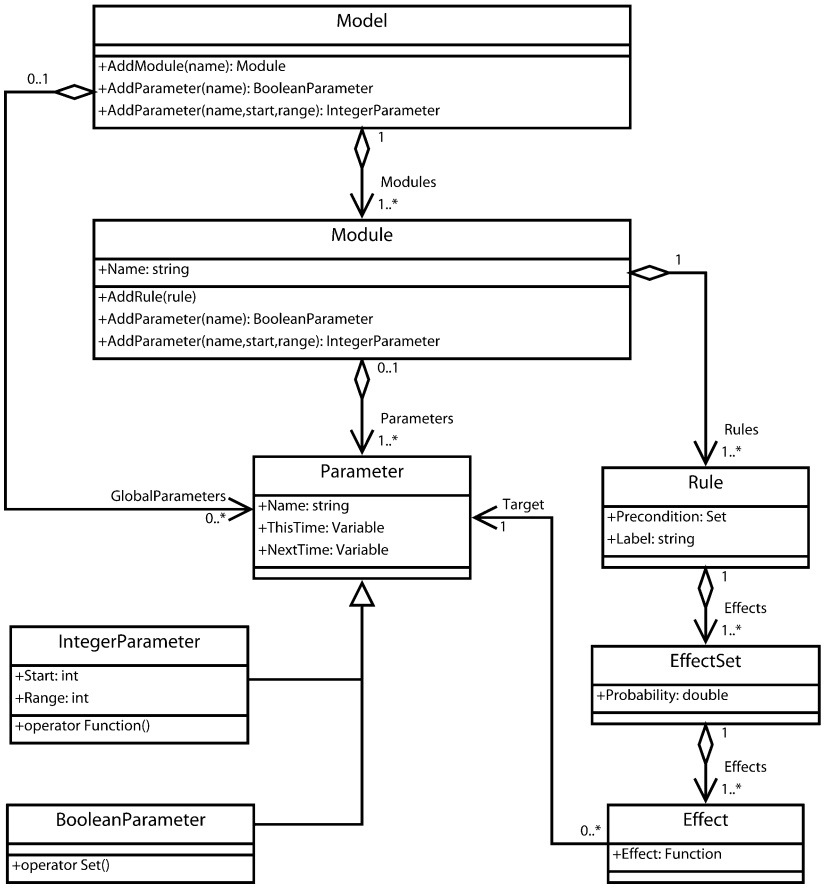
Предлагаемый нами инструмент вероятностной верификации использует для представления матрицы вероятностей переходов π многокорневые бинарные диаграммы решений, реализованные в рамках библиотеки BddFunctions [1]. Функционально инструмент представляет собой C++ библиотеку, предоставляющую объектно-ориентированный интерфейс, позволяющий сконструировать модель и проверить, выполнены ли относительно неё темпоральные свойства. Основные классы, формирующие интерфейс инструмента, представлены на рисунке.

Корневым классом является класс `Model`, обозначающий модель системы в целом. Статическая структура системы представлена в виде набора модулей (`Module`), каждый из которых содержит набор параметров (`Parameter`). На данный момент поддерживаются два типа параметров: флаги (`BooleanParameter`) и целые числа (`IntegerParameter`). Для модулей и параметров определены уникальные имена, причем имена параметров должны быть уникальны в контексте всей модели. Для целочисленных параметров, кроме того, определен диапазон допустимых значений $\{Start, \ldots, Start + Range\}$. Методы, используемые для инициализации модели, приведены в примере 1.

Поведение системы описывается в виде набора правил (`Rule`), ассоциированных с каждым модулем. Для правила может быть определено предусловие — множество состояний системы, в которых правило применимо, представленное классом `BddFunctions::Set` (часть библиотеки BddFunctions). Кроме того, правило задает множество наборов эффектов (`EffectSet`), для каждого из которых определена вероятность наступления. Сумма вероятностей наборов эффектов в рамках одного правила должна быть равна 1. Индивидуальные эффекты (`Effect`), входящие в набор, задают новое значение для некоторого параметра, потенциально зависящее от значений других параметров и представленное классом `BddFunctions::Function`. Не упомянутые в наборе эффектов параметры при применении правила сохраняют свои начальные значения. При определении предусловий и эффектов модуль может использовать любые параметры системы, однако применять эффекты он может только к своим собственным параметрам.

Правила добавляются с помощью метода `AddRule`, как показано в примере 2.

При совершении перехода система формирует набор модулей, в которых присутствует правило, предусловие которого выполнено в текущем состоянии (предусловия правил модуля не должны пересекаться, что гарантирует единственность правила для каждо-



Основные классы

Пример 1. Инициализация структуры модели

```
// Создаем модель (параметр определяет точность
// вычислений).
Model model(40);

// Добавляем модуль
Module module = model.AddModule("module1");

// Добавляем параметры
IntegerParameter x = module.AddParameter("x", 0, 256);
BooleanParameter a = module.AddParameter("a");

// Инициализируем библиотеку диаграмм решений
model.Initialize();
```

Пример 2. Определение правил

```
// Добавляем правило: "при наличии одного из флагов
// a b" с вероятностью 0.5 поменять их местами,
// с вероятностью 0.5 инвертировать b.
module1.AddRule(
  When(a | b)
    [0.5] >> (a = (Set)b, b = (Set)a)
    [0.5] >> (b = !b))

// Если не один из флагов не стоит, установить один
// из них случайным образом и увеличить
// соответствующий ему счетчик.
module1.AddRule(
  When(!a & !b)
    [0.5] >> (x = x + 1, b = !b)
    [0.5] >> (y = y + 1, a = !a));
```

го модуля), после чего случайным образом выбирает модуль, который совершит переход. Например, при наличии в системе двух модулей, в каждом из которых есть булевый флаг и правило вида $When(flag)\,[1] >> (flag = !flag)$, из состояния $(1,1)$ возможны два перехода: $(0,1)$ с вероятностью 0.5 (совершил переход первый модуль) и $(1,0)$ с вероятностью 0.5 (совершил переход второй модуль).

В некоторых случаях требуется синхронизация изменения состояний модулей. Одним из инструментов такой синхронизации являются *помеченные правила*. Если при определении правила задать метку, то это правило будет рассматриваться отдельно от остальных правил модуля совместно с другими правилами помеченны-

ми этой же меткой (условие непересекающихся предусловий в рамках модуля распространяется только на правила с одинаковыми метками). При обработке помеченных правил система для каждой метки формирует «виртуальный модуль», правила которого сформированы из помеченных правил участвующих модулей. Применение любого правила этого модуля приводит к тому, что все участвующие модули совершают переход одновременно. Если хотя бы один из модулей совершить переход не может (не выполнено предусловие), то вся группа исключается из рассмотрения. Например, если в рассмотренном выше примере с двумя правилами вида $When(flag)\,[1]\ >>\ (flag =!flag)$ добавить к обоим правилам одинаковую метку, то из состояния $(1,1)$ будет только один возможный переход: $(0,0)$ с вероятностью 1 (модули совершили переходы синхронно).

Еще одним средством синхронизации модулей являются глобальные параметры. Будучи определены на уровне модели в целом, они доступны каждому модулю и на чтение, и на запись. Однако менять значение глобальных параметров в помеченных правилах нельзя — это может привести к неоднозначности выбора значения.

После того как модель построена, вызов метода `Compile` приведет к построению в памяти диаграммы решений, представляющей матрицу вероятностей переходов системы. Для моделей некоторых классов после компиляции и перед началом верификации имеет смысл исключить из модели *недостижимые состояния* (состояния, в которые модель не сможет попасть из некоторого множества начальных состояний). Исключение недостижимых состояний, как правило, приводит к росту размера диаграммы решений (так как уменьшается регулярность модели), но оно может привести к ускорению сходимости вычислений, так как потенциальные флуктуации системы в недостижимых состояниях не учитываются. Для исключения недостижимых состояний необходимо вызвать метод `ApplyReachability`, передав множество начальных состояний в качестве параметра.

Для проведения верификации класс `Model` предоставляет следующие методы:

- `Function NextTime(const Set &target)` — строит для переданного множества состояний системы функцию, возвращающую для каждого состояния вероятность того, что из него

система на следующем ходу попадет в одно из состояний множества `target` (вычисляет $p_s(\bigcirc target)$);

- `Function Until(const Set & guard, const Set &target)` — определяет для каждого состояния системы вероятность когда-нибудь попасть в множество состояний `target`, пройдя только через состояния множества `guard` (вычисляет $p_s(guard\mathcal{U}target)$); аналогичный метод `MaxUntil` определяет *максимальную* вероятность, а метод `MinUntil` — *минимальную* вероятность[1];
- `Function Until(const Set &guard, const Set &target, uint k, void (callback*)(const Function&, size_t))` — для определения вероятности дойти до `target`, не выйдя из `guard` за `k` итераций (вычисляет $p_s(guard\mathcal{U}^k target)$, четвертый параметр метода является необязательным и может быть использован для обработки промежуточных результатов (для $k' = 1, \ldots, k$), этот метод также представлен в вариантах `MaxUntil` и `MinUntil`.

Заметим, что класс модели предоставляет только методы для вычисления формул путей. При этом для вычисления формул состояния используются операторы над множествами библиотеки BddFunctions, а для перевода формул пути в формулы состояния используются операторы сравнения функций. Использование этих методов для проверки свойства $P^{>=0.5}((P^{<=0.75} \bigcirc a)\mathcal{U}b\wedge!c)$ приведено в примере 3. Результатом работы этого кода является множество состояний, для которых выполнено искомое свойство (`satisfied`).

Пример 3. Проверка свойств

```
Set satisifed =
  model.MinUntil(
    model.NextTime(a) <= 0.75,
    b & !c) >= 0.5;
```

[1]Выбранный способ округления применяется не только к конечному результату, но и к промежуточным результатам — это является гарантией того, что полученные значения являются строгой нижней (верхней) границей оценки вероятности. При этом для оператора $\bigcirc target$ способ округления задавать не нужно, так как он вычисляется через сумму и не предполагает округления результатов.

Помимо рассмотренных выше возможностей, стоит обратить внимание еще и на несколько функций «тонкой» настройки работы инструмента. При создании экземпляра модели можно задать три параметра, регулирующие точность вычислений: количество бит, используемых в представлении модели (n)[2], количество бит, используемых при вычислениях (l), и количество бит, используемых для определения критерия завершения итеративного поиска решения при вычислении неограниченного оператора «когда-нибудь» (k). Рекомендованное отношение этих параметров: $n >= l > k$, а значения по умолчанию: $l = n$ и $k = l - 1$.

При инициализации модели с помощью метода `Initialize` существует возможность задать упорядочивание и группировку переменных для диаграмм решений, что, как известно, может существенно повлиять на размер диаграмм. Перегруженный вариант метода `Initialize` принимает в качестве параметра `list<list<string»`. Каждый список строк соответствует группе переменных, биты которых должны быть расположены в чередующем порядке. Сами же группы располагаются последовательно в соответствии с порядком в списке. В качестве имен выступают имена параметров для переменных текущего момента и имена параметров со знаком ' для переменных следующего момента (например, *a* и *a'*). По умолчанию в первую группу попадают все глобальные параметры, а остальные переменные группируются по модулям. Внутри групп переменные следуют попарно (текущий момент, следующий момент) в алфавитном порядке.

3. Экспериментальные результаты

Для экспериментального сравнения характеристик производительности и точности предложенного инструментария мы использовали два классических примера: выборы лидера в кольце процессоров и процесс рождений и смерти. В качестве эталона для сравнения используется популярный инструмент PRISM, основанный на многотерминальных диаграммах решений.

[2]Количество бит, равное n, говорит о том, что точность будет контролироваться вплоть до 2^{-n}.

3.1. Выборы лидера

Суть задачи сводится к тому, что в системе есть N процессоров, соединенных в кольцо. В некоторый момент времени процессорам нужно выбрать лидера. Для того чтобы это сделать каждый процессор выбирает случайное число из множества $\{1, \ldots, K\}$, после чего числа передаются по кругу N раз, чтобы каждый процессор увидел числа всех процессоров в кольце. Процессор, выбравший наибольшее уникальное число, становится лидером. Если выбрать лидера не удалось, то процесс повторяется. При верификации такой системы в первую очередь интересны два свойства: «лидер когда-нибудь будет выбран с вероятностью не меньше 1» ($P^{>=1} true \mathcal{U} elected$) и «лидер будет выбран с вероятностью не меньше p в не более чем L итераций» ($P^{>=p} true \mathcal{U}^{L} elected$).

В таблице 1 приведены время расчета и максимальный расход памяти при проверкt этих свойств для различных значений N, K при $L = 7$. Как видно из таблицы, для задач маленьких размеров ($N <= 4$) и PRISM, и наш инструментарий (MRBDD) дают схожие показатели производительности. Бóльший расход памяти PRISM обусловлен тем, что процесс выполняется в виртуальной машине Java, что требует некоторого константного количества памяти. Для средних задач ($(5, 6)$, $(5, 8)$ и $(6, 5)$) PRISM выигрывает по времени выполнения, до 50% при $N = 5$ и $K = 8$. При этом для самой сложной из рассмотренных задач ($N = 6$ и $K = 8$) время работы PRISM в 5 раз превзошло время работы нашего инструментария: 7 часов работы (25 118 сек) против 1,5 (5785 сек). Скорость роста расхода памяти у обоих инструментов сравнима, хотя здесь следует отметить, что большой расход памяти при использовании многокорневых диаграмм вызывает более агресивная политика кэширования, которая автоматически корректируется в условиях нехватки памяти.

Что же касается точности вычислений, то при проверке свойства $P^{>=1} true \mathcal{U} elected$ оба инструмента дали положительный ответ, оценив вероятность как точно равную 1. Результаты проверки свойства $P^{>=p} true \mathcal{U}^{L} elected$ приведены в таблице 2. В первом столбце таблицы дана оценка вероятности по результатам работы PRISM, тогда как вторые два столбца показывают верхнюю и нижнюю оценки вероятности, выданные нашим инструментом. Результаты PRISM

Таблица 1. **Выборы лидера (память в мегабайтах, время в секундах)**

N	*K*	*PRISM*		*MRBDD*	
		Memory	Time	Memory	Time
3	2	62,08	1	7,77	1
	3	62,29	2	7,86	1
	4	62,54	2	7,91	1
	5	62,72	2	8,31	2
	6	63,12	2	8,46	2
	8	63,43	2	9,05	2
4	2	62,28	2	7,83	1
	3	62,76	2	8,33	1
	4	63,79	3	9,15	4
	5	62,72	6	12,00	7
	6	68,73	8	14,28	7
	8	79,18	24	26,38	21
5	2	62,39	2	8,03	1
	3	64,66	3	10,10	4
	4	68,04	8	14,03	13
	5	79,67	30	33,02	33
	6	103,69	69	63,72	95
	8	215,13	285	167,68	581
6	2	62,96	2	8,38	3
	3	68,72	8	14,48	18
	4	87,82	44	44,82	77
	5	170,82	230	149,75	346
	6	375,41	1065	300,80	1083
	8	1193,38	25118	985,23	5785

получены с точностью до 15-го знака после запятой и попадают в «вилку», вычисленную нашим инструментом во всех случаях (как правило, практически у верхней границы). При этом видно, что точность вычисления «вилки» гораздо выше при K равном степени двойки (до 7-го знака после запятой), что объясняется тем, что при таких значениях происходит гораздо меньше округлений. При $K = 6$ точность вычисления «вилки» снижается до 5-го знака.

Таблица 2. **Выборы лидера (точность вычислений при N= 4)**

K	L	P_{PRISM}	P_{min}	P_{max}
4	1	0,8437500	0,8437499	0,8437500
	2	0,9755859	0,9755859	0,9755859
	3	0,9961853	0,9961852	0,9961853
	4	0,9994040	0,9994039	0,9994040
	5	0,9999069	0,9999068	0,9999069
	6	0,9999854	0,9999853	0,9999855
	7	0,9999977	0,9999976	0,9999977
6	1	0,9259259	0,9258985	0,9259701
	2	0,9945130	0,9944815	0,9945638
	3	0,9995936	0,9995616	0,9996450
	4	0,9999699	0,9999379	1,0000000
	5	0,9999978	0,9999657	1,0000000
	6	0,9999998	0,9999678	1,0000000
	7	1,0000000	0,9999680	1,0000000
8	1	0,9570313	0,9570312	0,9570313
	2	0,9981537	0,9981536	0,9981537
	3	0,9999207	0,9999206	0,9999207
	4	0,9999966	0,9999965	0,9999966
	5	0,9999999	0,9999998	0,9999999
	6	1,0000000	0,9999999	1,0000000
	7	1,0000000	0,9999999	1,0000000

По результатам этого эксперимента можно сделать вывод, что предложенный нами инструмент значительно превосходит PRISM по производительности при больших размерах задач, но уступает ему в точности при значениях K, отличных от степени двойки. Стоит также отметить, что в построенных моделях достаточно небольшое количество нетерминальных вершин, что лишает многокорневые диаграммы их основного преимущества по сравнению с многотерминальными.

3.2. Рождение и смерть

Процессы рождения и смерти используются для описания большого числа физических, биологических, информационных, экономических и социальных стохастических процессов, что делает востребованной задачу их анализа и прогнозирования. Математическая суть процесса заключается в том, что для некоторой *популяции* объектов определены законы появления (*рождения*) и исчезновения (*смерти*) элементов в зависимости от текущего размера популяции. Действуя одновременно, рождение и смерть могут влиять на популяцию сложно предсказуемым образом. Особый интерес в этом процессе представляют *точки невозвращения*. Например, при размере 0 (полное исчезновение) популяция уже никогда не сможет восстановиться.

Для эксперимента мы выбрали следующий закон рождений: для $i \in \{1, \ldots, 10\}$ с вероятностью $\frac{1}{2^i}$ количество рожденных будет равно $\frac{1}{2^{11-i}}$ доле текущей популяции, при этом с вероятностью $\frac{1}{2^{10}}$ рождений не будет вообще. Соответственно, наиболее вероятным является небольшой относительный прирост популяции. Для смерти закон определяется таким же образом, при этом рождение и смерть действуют совместно на каждом ходу. Основным параметром модели в данном случае будет число N, ограничивающее максимальный размер популяции следующим образом: $population < 2^N$.

С точки зрения верификации интересным является свойство «с какой вероятностью когда-нибудь будет достигнуто пороговое значение размера популяции x» ($P^{<=p} true \mathcal{U} population >= x$). Измерения производительности и расхода памяти для верификации приведены в таблице 3. Как видно из таблицы, на небольших задачах предложенный нами инструментарий показывает лучшее время работы, однако по мере роста N время работы обоих инструментов «взрываются», в результате чего верификация свойства занимает более 10 часов уже при $N = 9$ (т. е. популяция порядка 500 особей). Объясняется это тем, что итеративные методы решения для данного примера сходятся черезвычайно медленно (порядка 100 000 итераций), что делает верификацию для задач реалистичного размера черезвычайно трудоемкой.

В том случае, если не ограниченный по времени вариант свойства верифицировать не представляется возможным, интерес так-

Таблица 3. **Время работы и расход памяти при верификации примера «рождения и смерти»**

N	PRISM		MRBDD	
	Memory	Time	Memory	Time
3	62,00	3	7,84	1
4	66,05	6	9,89	3,8
5	62,75	65	17,71	28
6	68,03	392	46,84	130
7	65,52	1967	151,16	1275
8	97,41	9330	532,04	9360
9	197,45	37560	783,70	45936

Таблица 4. **Время работы и расход памяти при верификации ограниченного свойства**

N	PRISM		MRBDD	
	Memory	Time	Memory	Time
3	61,72	1	7,74	1
4	61,63	1	7,93	1
5	61,95	2	11,56	2
6	63,24	3	11,62	12
7	65,22	11	18,36	22
8	92,98	75	30,70	30
9	197,17	628	46,29	114
10	357,88	3633	66,04	256
11	1222,36	39555	104,18	494
12			108,22	805
13			112,64	1677
14			124,13	3630
15			180,22	6549
16			261,55	19278

же представляет его ограниченный по времени аналог: «с какой вероятностью не более чем через L итераций будет достигнуто пороговое значение размера популяции x» ($P^{<=p} true \mathcal{U}^L population >= x$). Результаты верификации для этого свойства приведены в таблице 4. При ограничении числа итераций для PRISM ситуация принципиально не меняется — при $N = 11$ (популяция порядка 2000 особей) расчет занимает более 10 часов при расходе памяти более 1 бай-

та. Предлагаемый же нами инструментарий показывает гораздо более привлекательные результаты: верификацию при $N = 16$ (что соответствует популяции в 65 000 особей) можно провести за 5 часов при расходе памяти в 260 байт (что говорит о том, что расчет возможен и для больших N).

Заключение

В данной работе предлагается инструментарий вероятностной верификации, основанный на многокорневых бинарных диаграммах решений и реализованный в виде подключаемой библиотеки для языка C++. Выбор этого формата обусловлен в первую очередь наибольшей гибкостью при формировании моделей, а также возможностью использовать инструмент не только для непосредственной верификации, но и для решения других задач, где верификация является только одним из этапов.

Экспериментальное сравнение с таким популярным инструментом вероятностной верификации, как PRISM, показывает, что для большинства задач предложенный нами инструментарий показывает аналогичные результаты по производительности, однако для задач большого размера время работы предложенного инструментария может быть в разы меньше времени работы PRISM. В частности, при верификации процессов рождения и смерти предложенный инструментарий позволяет обрабатывать задачи существенно бóльшего размера.

Следует отметить, что в предложенном инструментарии присутствует достаточно большой потенциал для оптимизации времени работы. Если основанные на многотерминальных диаграммах решений инструменты (в том числе PRISM) при выполнении операций с числами используют стандартные операции процессора, прошедшие более чем полувековую эволюцию, то инструмент, основанный на многокорневых диаграммах, использует достаточно простые алгоритмы арифметики с неподвижной точкой. Таким образом, актуальной является задача разработки оптимизированных алгоритмов вычисления арифметических операций для работы с многокорневыми диаграммами решений. Кроме того, одним из интересных направлений для развития предложенного подхода является внедрение параллельных и распределенных вычислений.

Интересным направлением дальнейшего развитие может быть и реализация инструментов визуализации диаграмм решений и их фрагментов, что позволит проанализировать их структуру и оптимизировать кодировку и упорядочивание переменных. Для реализации инструментария визуализации можно использовать, например, DSM-подход [4].

Список литературы

[1] *Бугайченко Д. Ю., Соловьев И. П.* Библиотека многокорневых бинарных решающих диаграмм BddFunctions и ее применение // Системное программирование. Вып. 5. СПб.: Изд. СПбГУ, 2010.

[2] *Бугайченко Д. Ю., Соловьев И. П.* Моделирование дискретных случайных величин на основе многокорневых диаграмм решений // Сборник трудов XVI международной конференции «Современные проблемы информатизации в экономике и обеспечении безопасности». 2011. С. 63–70.

[3] *Карпов Ю. Г.* Model Checking. Верификация параллельных и распределенных программных систем. БХВ-Петербург, 2010. 552 с.

[4] *Кознов Д. В.* Разработка и сопровождение DSM-решений на основе MSF // Системное программирование. Вып. 3 / под ред. А. Н. Терехова и Д. Ю. Булычева. СПб.: Изд. СПбГУ, 2008. С. 80–96.

[5] *Bäumler S., Balser M., Dunets A.* et. al. Verification of Medical Guidelines by Model Checking-a Case Study // Model Checking Software. 2006. P. 219–233.

[6] *Beyer D., Henzinger T., Jhala R., Majumdar R.* The Software Model Checker Blast // International Journal on Software Tools for Technology Transfer (STTT). Vol. 9, N 5. 2007. P. 505–525.

[7] *Bryant R. E.* Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams // ACM Computing Surveys. 1992. Vol. 24, N 3. P. 293–318.

[8] *Bugaychenko D., Soloviev I.* Application of Multiroot Decision Diagrams for Integer Functions // Vestnik St. Petersburg University: Mathematics. 2010. Vol. 43, N 2. P. 92–97.

[9] *Constant C., Jeannet B., Jéron T.* Automatic Test Generation from Interprocedural Specifications // Testing of Software and Communicating Systems. 2007. P. 41–57.

[10] *Duff I.* A Survey of Sparse Matrix Research // Proceedings of the IEEE. 2005. Vol. 65, N 4. P. 500–535.

[11] *Itai A., Rodeh M.* Symmetry Breaking in Distributed Networks // Information and Computation. 1990. Vol. 88, N 1.

[12] *Fujita M., McGeer P. C., Yang J. C.-Y.* Multi-Terminal Binary Decision Diagrams: An Efficient Datastructure for Matrix Representation // Form. Methods Syst. Des. 1997. Vol. 10, N 2–3. P. 149–169.

[13] *Kong S., Tillmann N., Halleux J. d.* Automated Testing of Environment-Dependent Programs — a Case Study of Modeling the File System for Pex // ITNG '09: Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations. Washington, DC, USA: IEEE Computer Society, 2009. P. 758–762.

[14] *Oldenkamp H.* Probabilistic Model Checking. A Comparison of Tools. Master's Thesis. University of Twente, Niederlande, 2007.

[15] *Pang J., Fokkink W., Hofman R., Veldema Model R.* Checking a Cache Coherence Protocol of a Java DSM Implementation / // Journal of Logic and Algebraic Programming. 2007. Vol. 71, N 1. P. 1–43.

[16] *Parker D.* Implementation of Symbolic Model Checking for Probabilistic System: Ph.D. thesis / University of Birmingham. 2002.

[17] *Resnick S.* Adventures in Stochastic Processes. The Random World of Happy Harry. Birkhäuser, 1992. 626 p. URL: http://books.google.com/books?id=YGjTl8iX-HsC

[18] *Tillmann N., De Halleux J.* Pex: White Box Test Generation for .NET // TAP'08: Proceedings of the 2nd International Conference on Tests and Proofs. Berlin; Heidelberg: Springer-Verlag, 2008. P. 134–153.

[19] *Younes H., Kwiatkowska M., Norman G., Parker D.* Numerical vs. Statistical Probabilistic Model Checking // International Journal on Software Tools for Technology Transfer (STTT). 2006. Vol. 8, N 3. P. 216–228.

[20] *Zaki M., Tahar S., Bois G.* Formal Verification of Analog and Mixed Signal Designs: A Survey // Microelectronics Journal. 2008. Vol. 39, N 12. P. 1395–1404.