

DOI: 10.15514/ISPRAS-2026-38(6)-3



Подход к статической верификации многопоточных приложений на основе графовых моделей программы

О.С. Крюков, ORCID: 0000-0003-3155-6266 <ol_kryukov97@mail.ru>

А.Г. Волошко, ORCID: 0000-0002-4304-2513 <atroshina@mail.ru>

А.Н. Ивутин, ORCID: 0000-0003-2970-2148 <alexey.ivutin@gmail.com>

Тульский государственный университет,
Россия, 300012, г. Тула, проспект Ленина, д. 92.

Аннотация. Работа посвящена разработке метода статической верификации параллельных программ на основе анализа автоматически построенных формальных моделей программ. В качестве решения предложен графовый подход на основе расширенной сети Петри с семантическими связями, которая за счет явного разделения потоков управления и данных в рамках двудольного орграфа позволяет эффективно выявлять критические ошибки параллелизма, такие как гонки данных и взаимные блокировки. Практическая значимость подтверждена прототипом Semantic Petri Net Analyzer для языка Go, который в ходе эксперимента на наборе данных GoBench показал способность обнаруживать гонки данных и взаимные блокировки, продемонстрировав преимущество перед динамическими методами. Показано, что предложенный метод обеспечивает сбалансированное решение для повышения надежности параллельного программного обеспечения, сочетая полноту анализа, автоматизацию и практическую применимость.

Ключевые слова: статическая верификация; параллельные программы; сеть Петри; состояние гонки; взаимная блокировка; анализ модели.

Для цитирования: Крюков О.С., Волошко А.Г., Ивутин А.Н. Подход к статической верификации многопоточных приложений на основе графовых моделей программы. Труды ИСП РАН, том 38, вып. 6, часть 1, 2026 г., стр. 41–62. DOI: 10.15514/ISPRAS-2026-38(6)-3.

Approach to static verification of multithreaded applications based on a graph model of a program

O.S. Kryukov, ORCID: 0000-0003-3155-6266 <ol_kryukov97@mail.ru>

A.G. Voloshko, ORCID: 0000-0002-4304-2513 <atroshina@mail.ru>

A.N. Ivutin, ORCID: 0000-0003-2970-2148 <alexey.ivutin@gmail.com>

Tula State University,
92, Prospekt Lenina, Tula, 300012, Russia.

Abstract. This work is devoted to developing a static verification method for concurrent programs through the analysis of automatically constructed formal program models. The proposed solution is a graph-based approach utilizing an extended Petri net with semantic relations, which, owing to the explicit separation of control and data flows within a directed bipartite graph, enables efficient detection of critical concurrency errors, such as data races and deadlocks. The practical significance of the approach is confirmed by a prototype implementation – the Semantic Petri Net Analyzer for the Go language – which, in experiments conducted on the GoBench benchmark, demonstrated its ability to detect data races and deadlocks and outperformed dynamic analysis methods. It is shown that the proposed method offers a balanced solution for enhancing the reliability of concurrent software, combining analysis completeness, automation, and practical applicability.

Keywords: static verification; concurrent programs; Petri net; data race; deadlock; model checking.

For citation: Kryukov O.S., Voloshko A.G., Ivutin A.N. Approach to static verification of multithreaded applications based on a graph model of a program. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 6, part 1, 2026, pp. 41–62 (in Russian). DOI: 10.15514/ISPRAS-2026-38(6)-3.

1. Введение

Создание параллельного программного обеспечения сопряжено с существенными методологическими трудностями, обусловленными недетерминированной природой конкурентных вычислений: множество потоков управления (включая потоки операционной системы, легковесные единицы выполнения и процессы) функционирует асинхронно и конкурентно, вступая в борьбу за разделяемые аппаратные и программные ресурсы. Данное обстоятельство принципиально трансформирует процесс проектирования, когда разработчик вынужден оперировать категориями независимых сущностей, их взаимодействия и синхронизации, отказываясь от привычной модели последовательного алгоритма. Необходимость учёта комбинаторно значительного числа возможных сценариев чередования операций в потоках существенно повышает трудоёмкость отладки и создаёт предпосылки для возникновения трудновоспроизводимых ошибок.

Отличительной причиной нестабильности поведения параллельной программы выступает наличие гонки данных (Data Race) или состояния гонки (Race Condition), требующее для обеспечения детерминированного поведения применения специализированных механизмов синхронизации: примитивы взаимного исключения, семафоры, каналные примитивы и др. Тем не менее, внедрение этих инструментов не является панацеей, поскольку порождает новые классы программных ошибок, связанных уже с некорректной синхронизацией: взаимные блокировки (Deadlocks), голодание (Starvation), инверсия приоритетов (Priority Inversion).

Специфика ошибок параллельного программирования приводит к фундаментальной проблеме – сложности их гарантированного обнаружения. Недетерминированность выполнения делает классический итеративный подход «запусти-проверь-исправь» малоэффективным, поскольку ошибка может не воспроизводиться.

Существующие подходы к верификации имеют принципиальные ограничения. Динамические методы [1–3] не только вносят высокие накладные расходы, но и не обеспечивают полного покрытия пространства состояний, кратно расширяющегося в

сравнении с последовательным кодом. Статическая верификация не требует запуска кода [1,4-5], может применяться до этапа тестирования и теоретически способна охватить все возможные траектории выполнения, однако, для большинства таких методов ключевой проблемой становится комбинаторный взрыв числа состояний даже для несложных параллельных программ. Для его смягчения используются техники абстракции, заменяющие бесконечное пространство состояний конечным набором абстрактных элементов [6]. Второй не менее важной проблемой является требование к построению формальной модели программы [5] заданного уровня абстракции, что в ряде случаев требует глубоких знаний математической логики и значительного участия человека, затрудняя полную автоматизацию.

Таким образом, совершенствование методов статической верификации, особенно автоматического построения модели программы, остаётся актуальной задачей. В работе авторы предлагают новый подход к статической верификации, решающий одну из ключевых проблем систем данного класса – автоматизацию построения модели программы, максимально приближенной к реальному коду, что позволяет существенно повысить точность детектирования наиболее распространённых ошибок параллельного программирования (состояния гонки (гонки данных) и взаимные блокировки (Deadlocks)) по сравнению с решениями на основе приближенных моделей, у которых сохраняется проблема полного покрытия пространства состояний, или динамических верификаторов, для которых дополнительно снимается проблема ошибок детектирования первого и второго рода.

2. Выбор модели параллельной программы

Ключевой задачей становится выбор адекватной формальной модели параллельной программы как с точки зрения ее пригодности для автоматического анализа и обнаружения целевых ошибок (гонки данных, взаимные блокировки), так и с точки зрения возможности ее автоматического построения из параллельного кода с учетом используемых в нем механизмов распараллеливания и синхронизации. Такая модель должна явно представлять одновременное выполнение нескольких процессов, учитывать зависимости как по данным (фиксировать, как операции используют результаты других операций), так и по управлению (отображать последовательность выполнения операций внутри одного процесса).

Существующие формальные модели представления параллельных программ могут быть разделены на два принципиально различных класса: аналитические (логико-алгебраические) и графовые (исполнимые) модели.

Аналитические модели описывают поведение вычислительной системы посредством логических формул, аксиом и утверждений. К данному классу относятся разнообразные формальные исчисления [7]: исчисление высказываний, исчисление предикатов, λ -исчисление и их расширения. Применение таких моделей ограничено в основном дедуктивным анализом, где они могут использоваться благодаря высочайшей степени строгости, но построение подобных спецификаций остаётся наименее автоматизируемым этапом анализа и требует значительного участия человека в формулировании утверждений, инвариантов и доказательств. Поэтому в дальнейшем не будем рассматривать данный вид моделей для нашей задачи.

Альтернативой в рамках аналитического класса выступает применение темпоральных логик. Характерным примером служит язык формальных спецификаций TLA+ (Temporal Logic of Actions) [8], основанный на темпоральной логике действий. С его помощью формулируются свойства, которым должна удовлетворять распределённая или параллельная система (например, «инвариант не нарушается никогда» или «событие A всегда предшествует событию B»). TLA+ является мощным инструментом для анализа выполнимости и верификации алгоритмов на уровне высокоуровневой спецификации. Вместе с тем, автоматическое построение TLA+-модели непосредственно из исходного кода программ

сопряжено с фундаментальными трудностями: темпоральные утверждения оперируют абстрактными предикатами над состояниями, тогда как программный код содержит множество низкоуровневых деталей реализации (управление памятью, синтаксические конструкции синхронизации, особенности планирования потоков), семантическая редукция которых к компактной логической спецификации не поддаётся полной автоматизации, а написание корректной спецификации разработчиком требует от него глубокой экспертизы в области формальных методов и приводит к значительным временным затратам. Таким образом, данный класс моделей также исключается из дальнейшего рассмотрения.

Графовые модели представляют программу в виде ориентированного графа [9], вершины которого соответствуют состояниям или операциям, а дуги – переходам либо информационным зависимостям между ними. Основное достоинство графового подхода заключается в его структурной близости к внутреннему представлению программы в компиляторах и анализаторах кода, что делает возможным полностью автоматическое построение модели из исходных текстов. Кроме того, графовая форма представления опирается на обширный математический аппарат теории графов, для которого разработано множество эффективных алгоритмов поиска путей, циклов, компонент связности и достижимости, что создаёт предпосылки для практической реализации масштабируемых инструментов статической верификации.

К основным вариантам графовых моделей, применяемым при анализе параллельных программ, относятся: конечные автоматы, граф потока управления, граф «операции-операнды» (потока данных) и сети Петри.

Конечные автоматы моделируют систему как совокупность дискретных состояний и переходов между ними. При анализе параллельных систем применяются композиции автоматов (системы конечных автоматов с параллельными процессами) [10]. Существенным ограничением данного представления является то, что даже в композиционной форме модель слабо отражает истинный параллелизм и подвержена комбинаторному взрыву числа состояний при попытке явного перебора всех допустимых комбинаций состояний взаимодействующих компонентов.

Граф потока управления (Control Flow Graph, CFG) отображает последовательность выполнения базовых блоков кода в рамках отдельного потока управления. Данное представление эффективно используется для анализа покрытия кода, обнаружения недостижимых ветвей и оптимизации последовательных участков программы. Однако CFG не учитывает зависимости по данным и не предназначен для непосредственного моделирования межпоточного взаимодействия, синхронизации и конкурентного доступа к разделяемым ресурсам.

Граф «операции-операнды», или граф потока данных (Data Flow Graph, DFG), отражает информационные зависимости между операциями, показывая, каким образом результаты вычислений передаются от одной операции к другой в качестве операндов. Данная модель позволяет выявить потенциальный параллелизм на уровне отдельных инструкций или выражений. Тем не менее, сама по себе модель потока данных не специфицирует последовательность и условия выполнения операций, что ограничивает её применимость для анализа синхронизационных аспектов параллельного исполнения, где критически важны как порядок, так и контекст межпоточного взаимодействия.

Сети Петри являются математическим аппаратом, изначально предназначенным для моделирования параллельных, асинхронных, распределённых и недетерминированных систем [11-12]. За более чем полувековую историю они нашли широкое применение как в производственных процессах, так и в программной инженерии [13], в том числе для моделирования многопоточных приложений [14]. В различных областях сети Петри, и в частности их расширения, находили применение для поиска ошибок в проектируемых и реализуемых процессах. Так, для поиска тупиковых ситуаций в автоматизированных

производственных системах широко используются сети класса S3PR и S4PR [15-16]. Для поиска ошибок зависимостей по данным в системах некоторые исследователи предлагают использовать сети PN-DO [17]. Существенный прогресс в исследованиях был достигнут с введением класса сетей Гадара, применяемых для моделирования многопоточных программ с разделяемой памятью и операциями захвата/освобождения блокировок. Для этих сетей разработаны эффективные алгоритмы верификации живости, обратимости и синтеза оптимального управления, предотвращающего взаимоблокировки [18-20]. Развитием теории сетей Петри в области автоматизации обнаружения ошибок в многопоточных системах является работа Sun и соавторов [21]. Отдельно отметим работу [22] в области автоматического построения модели программы в терминах сетей Петри с использованием редукции и композиционного представления. К сожалению, работ в области сетей Петри и их применения для моделирования программ слишком много, чтобы упомянуть их все в рамках данной статьи, но можно сделать вывод, что сети Петри и их расширения зарекомендовали себя как эффективный формальный аппарат для анализа параллельных программ, что подтверждает обоснованность их выбора в качестве основы для предлагаемого подхода.

Исходя из этого наилучшим решением представляется выполнить разделение этих сущностей, сформировав в сети Петри два класса переходов – по управлению (моделируют последовательность выполнения операций, ветвление, порождение параллельных потоков и их синхронизацию, отвечают за «когда» и «в каком порядке» выполняются операции) и по семантическим связям (моделируют зависимости по данным и явно показывают, какая операция производит данные, а какая их потребляет). Получаемая в результате расширенная сеть Петри с семантическими связями (РСПСС) позволяет независимо, но в рамках единой целостной модели, анализировать поток управления и поток данных, что повышает наглядность и упрощает алгоритмы поиска ошибок. В РСПСС наличие нескольких переходов по семантике, связывающих одну позицию-переменную с позициями в параллельных частях модели, является прямым индикатором состояния гонки, что упрощает формальные правила поиска ошибок такого типа. Расширение сети вводит типизацию переходов, что позволяет точно отражать такие операции, как чтение данных из нескольких источников или использование одного результата в нескольких операциях, что критично для поиска ошибок. Разделение сети на две подсети позволяет применять более целенаправленные и эффективные алгоритмы верификации для каждого типа ошибок (для поиска взаимных блокировок используется подсеть по управлению, а для поиска гонок данных – семантическая подсеть).

Таким образом, классические сети Петри были выбраны как наиболее адекватный вариант представления благодаря своей ориентации на параллелизм. Однако для преодоления их ограничений и точного моделирования программной семантики было разработано собственное расширение – РСПСС. Это расширение обеспечивает необходимый уровень детализации для надежного автоматического обнаружения как ошибок синхронизации (взаимных блокировок), так и ошибок конкурентного доступа к данным (гонки данных), что делает его оптимальной моделью для построения целевого средства статической верификации. Рассмотрим математический аппарат РСПСС более подробно.

3. Описание модели параллельной программы

Расширенная сеть Петри с семантическими связями – это формальная модель, специально разработанная для представления и анализа параллельных процессов [23,4]. Её ключевая особенность – разделение потоков управления и данных, что позволяет точно моделировать и обнаруживать специфические ошибки параллелизма. Структура сети задается следующим множеством:

$$\Pi = \{A, \{Z^C, \bar{R}^C, \bar{R}^S\}, \{Z^S, \bar{R}^S, \bar{R}^S\}\},$$

где A – конечное множество позиций; Z^C – конечное множество переходов по управлению; Z^S – конечное множество переходов по семантическим связям; $\bar{R}^C$ – матрица смежности, отображающая множество позиций в множество переходов по управлению; $\bar{R}^C$ – матрица смежности, отображающая множество переходов по управлению в множество позиций; $\bar{R}^S$ – матрица смежности, отображающая множество позиций в множество переходов по семантическим связям; $\bar{R}^S$ – матрица смежности, отображающая множество переходов по семантическим связям в множество позиций.

Таким образом, РСПСС представляет собой трехдольный орграф с вершинами следующих типов: позиции, служащие для моделирования операций программы (вычисления, чтение, запись) или состояния данных (переменные, память), переходы по управлению для представления порядка выполнения операций и переходы по семантическим связям для представления зависимости по данным. Использование различных типов переходов по управлению позволяет моделировать параллельность исполнения. Для создания параллельных потоков используется переход по управлению типа fork, для объединения потоков – join, для синхронизации потоков посредством барьеров – synchro. Для представления последовательного исполнения кода используются примитивные переходы. Переходы по семантическим связям также бывают различных типов: примитивные моделируют передачу данных между двумя операциями, s-join – сбор данных из нескольких источников. Указанные переходы отличаются множествами входных ($I_A(Z^S), I_A(Z^C)$) и выходных позиций ($O_A(Z^S), O_A(Z^C)$).

Графически позиции сети представляются кругами, переходы по управлению – утолщенными линиями, переходы по семантическим связям – треугольниками. Каждая позиция РСПСС имеет не более одной дуги, связывающей ее с любым переходом. При этом из позиции может выходить несколько дуг, ведущих к разным переходам в случае ветвления, и входить несколько дуг при завершении альтернативных путей алгоритма.

В связи с добавлением еще одного типа переходов в РСПСС фишка представляет собой сложную структуру, состоящую из семантической и управляющей части, каждая из которых взаимодействует со своим типом переходов, объединяясь в позиции. Проверка возможности объединения фишки в позиции позволяет дополнительно отслеживать ошибки в порядке следования операций с точки зрения информационных зависимостей, если в позицию приходит только управляющая фишка без семантической. В любой позиции сети кроме позиций блокировки в любой момент времени не может быть более одной объединенной фишки. Позиции блокировки могут содержать несколько фишек для их передачи в несколько одновременно исполняемых потоков.

Начальная разметка в РСПСС, моделирующей параллельную программу, предполагает наличие одной объединенной фишки в стартовой позиции и необходимого числа фишек в позициях блокировки.

В качестве примера рассмотрим программу на языке Go с двумя потоками, обращающимися к общей переменной:

Представленная программа представляет собой оперирование значениями переменной x в трех потоках – главном и двух дочерних. РСПСС для данной программы представлена на рис. 1. Для каждой операции программы в РСПСС имеется отдельная позиция: 0 – выделение памяти для переменной x , 1 – оператор go для порождения первого потока, 2 – операция записи константы 10 в переменную x в первом потоке, 3 – оператор go для порождения второго потока, 4 – операция записи константы 20 в переменную x в главном потоке, 5 – выделение памяти для переменной y , 6 – операция чтения значения переменной x из памяти, 7 - операция записи считанного значения переменной x в переменную y , 11 – терминатор функции.

Соответственно переходы по управлению отображают процесс передачи управления от одной операции к следующей (следующим), например, переходы по управлению 1 и 2

показывают передачу управления в несколько параллельных потоков после порождения нового. Переходы по семантическим связям в свою очередь отражают передачу данных от одной операции к другой: 0 и 2 – запись в конкретную память, 1, 3 и 6 – зависимость памяти от операции записи в неё, 4 – чтение памяти переменной x , 5 – использование считанного значения переменной x для записи в памяти переменной y .

```

01 var x int
02 // Поток 1
03 go func() {
04     x = 10
05 }()
06 // Поток 2
07 go func() {
08     y := x
09 }()
10 x = 20
    
```

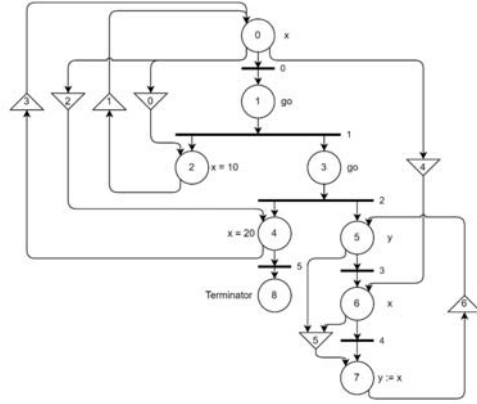


Рис. 1. Пример PCFG для параллельной программы.
Fig. 1. Example of ExPNSR for the parallel program.

Таким образом, модель PCFG обеспечивает высокую степень соответствия семантике параллельного выполнения, сохраняя при этом необходимый уровень формальной строгости и допуская возможность автоматизированного построения и анализа. Совокупность перечисленных свойств позволяет рассматривать её в качестве перспективной основы для инструментальных средств верификации параллельных программ.

4. Метод поиска взаимных блокировок в параллельных программах

Метод статической верификации параллельных программ для поиска взаимных блокировок (тупиков) основан на анализе модели программы, представленной в виде PCFG. Ключевая идея заключается в том, что для обнаружения тупиков достаточно анализировать только подсеть по управлению, которая моделирует последовательность исполнения команд и распределение операций по потокам.

Процесс анализа делится на два основных этапа, соответствующих двум распространенным типам тупиков:

- 1) анализ порядка захвата блокировок (тупик 1-го типа);
- 2) анализ отсутствия освобождения блокировок (тупик 2-го типа).

Основой для анализа является идентификация в модели позиций блокировки (семафоров, примитивов взаимного исключения) и параллельных путей выполнения.

Таким образом, можно выделить основные этапы метода, необходимые для поиска взаимных блокировок:

- 1) идентификация параллельных областей и входящих в них параллельных путей по управлению;
- 2) определение позиций блокировки;
- 3) анализ порядка взятия блокировок;
- 4) анализ отсутствия освобождения блокировки.

Рассмотрим подробнее правила и определения, связанные с данными шагами.

4.1 Идентификация параллельных областей и входящих в них параллельных путей по управлению

Прежде чем переходить к поиску тупиковых ситуаций, необходимо выделить в модели те фрагменты, в которых возможно возникновение конкурентного взаимодействия.

Определение 1. Путем по управлению называется ориентированный граф $P^C = (A, Z^C, E)$, где A, Z^C – вершины, E – ребра, причем

$$\forall e = \{u, v\} \in E: (u \in A \wedge v \in Z^C) \vee (u \in Z^C \wedge v \in A).$$

Определение 2. Позиция a_i считается предшествующей по управлению для a_j ($a_i <_C a_j$), если

$$a_i <_C a_j \Leftrightarrow \exists P_k \in P_C(a_{start}, a_j) (a_i \in P_k),$$

где $P_C(a_{start}, a_j)$ – множество всех путей по управлению из a_{start} в a_j

Определение 3. Параллельная область $\Omega = \{P_i\}$ представляет собой множество путей выполнения, такое, что

$$\Omega = \{P_i \in P_\Omega \mid \text{Bound}(\Omega) \subseteq \{\text{Fork}, \text{Join}, \text{Synchro}\} \wedge \text{Block}(P_i)\}$$

$$\text{Nested}(\Omega_c, \Omega_p) \Leftrightarrow \Omega_c \subseteq \Omega_p.$$

Определение 4. Два пути P_i и P_j в PCFG являются параллельными ($P_i \parallel_C P_j$), если они принадлежат одной параллельной области и содержат несовместимые по управлению позиции. Это означает, что не существует такого пути от стартовой позиции, который проходил бы через позиции из обоих путей: $a_i \not\prec_C a_j \Leftrightarrow a_i \prec_C a_j \wedge a_j \prec_C a_i$.

Выделение параллельных областей и установление отношения параллельности между путями выполнения формирует структурную основу для дальнейшего анализа ошибок синхронизации. Следует отметить, что наличие в модели нескольких параллельных путей само по себе не является достаточным условием возникновения тупика – для этого необходимо также присутствие элементов, моделирующих примитивы синхронизации.

4.2 Определение позиций блокировки

Представленная система параллельных путей и областей позволяет установить, какие фрагменты программы могут исполняться конкурентно, однако это условие не является достаточным, и для анализа корректности межпоточного взаимодействия необходимо также специфицировать те компоненты модели, которые отвечают за синхронизацию доступа к общим данным. С этой целью вводятся понятия, формализующие примитивы синхронизации в терминах PCFG.

Определение 5. Позиция блокировки – некоторая позиция сети, представляющая собой состояние примитива синхронизации.

Позиция a_m считается позицией блокировки примитива взаимного исключения, если

$$\text{Mutex}(a_m) \Leftrightarrow \exists P_i, P_j \in \Omega, i \neq j, \exists z_p^C \in P_i, \exists z_q^C \in P_j: \hat{R}_{p,m}^C = 1 \wedge \hat{R}_{m,q}^C = 1 \wedge M_0(a_m) = 1$$

Это позволяет формально описывать захват (a_m является входом перехода z^C) и освобождение (a_m является выходом перехода z^C) блокировки разными потоками. Убрав ограничение на количество фишек на начальной разметке, можно получить описание позиции блокировки семафора.

$$\exists(z_i^C \in Z^C) \wedge \exists(z_j^C \in Z^C) \wedge \exists(a_m \in A) \wedge a_m \notin P_i \wedge a_m \notin P_j \wedge a_m \in I_{A(z_i^C)} \wedge$$

$$a_m \in O_A(z_j^C) \wedge |I_A(z_i^C)| = |O_A(z_j^C)| \wedge z_i^C \in P_i \wedge z_j^C \in P_j \wedge z_j^C \notin P_i \wedge$$

$$z_i^C \notin P_j \wedge P_i \parallel P_j \wedge \forall(n \neq i \wedge n \neq j) a_m \notin I_A(z_n^C) \wedge a_m \notin O_A(z_n^C) \rightarrow \text{semaphore}(a_m).$$

Прочие примитивы синхронизации нет необходимости обнаруживать отдельным образом, так как их либо возможно представлять стандартными средствами РСПСС [24] (атомарная операция – отдельная позиция, барьер – переход по управлению типа Synchro), либо основаны на их применении (условная переменная, монитор и др.) и требуют только специального представления в модели, но не специального анализа.

Исходя из наличия в модели позиции блокировки, возможен её анализ на наличие взаимоблокировок (тупиков).

4.3 Правила обнаружения взаимных блокировок

Взаимные блокировки (тупики) в параллельной программе можно определить следующим образом.

- 1) Тупик – наличие в параллельной области более одной блокировки, причем последовательность обращения к позиции блокировки из различных участков области отличается.
- 2) Тупик – отсутствие разблокировки, то есть поток, взявший блокировку, не освобождает её.

Как видно из определения, условием возникновения тупика первого типа является наличие в параллельной области Ω как минимум двух позиций блокировки a_i и a_j . Тупик возникает, если в параллельных путях P_i и P_j существует циклическая зависимость по захвату этих блокировок: путь P_i сначала захватывает a_i , а затем пытается захватить a_j , путь P_j сначала захватывает a_j , а затем пытается захватить a_i . Таким образом, для обнаружения некорректного порядка захвата блокировок можно использовать следующее правило:

$$L \subset A(\Omega) \wedge |L| > 1 \wedge \exists(a_i, a_j \in L) \wedge \exists(a_k, a_l \in P_i) \wedge \exists(a_m, a_n \in P_j) \wedge a_k <_C a_l \wedge a_m <_C a_n \wedge \exists(z_k^C \in Z_C) \wedge a_i \in I_A(z_k^C) \wedge a_k \in O_A(z_k^C) \wedge \exists(z_l^C \in Z_C) \wedge a_j \in I_A(z_l^C) \wedge a_l \in O_A(z_l^C) \wedge \exists(z_m^C \in Z_C) \wedge a_j \in I_A(z_m^C) \wedge a_m \in O_A(z_m^C) \wedge \exists(z_n^C \in Z_C) \wedge a_i \in I_A(z_n^C) \wedge a_n \in O_A(z_n^C) \rightarrow \text{deadlock}(\Omega),$$

где L – множество позиций блокировки.

Поиск взаимных блокировок, согласно описанному правилу, выполняется следующим образом: в двух параллельных путях отыскиваются все позиции взятия блокировки, после чего проверяется, сохраняется ли между ними отношение предшествования по управлению в зависимости от того, какая блокировка взята. Пути по управлению, содержащие менее двух операций взятия блокировки, считаются свободными от взаимной блокировки. Позиция взятия блокировки определяется как выход перехода по управлению, на входе которого находится позиция блокировки. На рис. 2 представлена РСПСС, соответствующая описанному правилу.

Во втором случае взаимоблокировка связана с отсутствием разблокировки и может встречаться как в той же параллельной области Ω_1 , так и в последующей Ω_2 . Считается, что тупик присутствует в программе, если хотя бы в одном из параллельных путей, взявших блокировку, отсутствует операция, ведущая к освобождению этой блокировки (то есть нет перехода, для которого a_i , являющаяся позицией блокировки, является выходом). Кроме того, если в Ω_1 наблюдается тупиковая ситуация, связанная с a_i , то она также будет присутствовать и в Ω_2 . Это можно описать следующими правилами:

$$L \subset A(\Omega) \wedge \exists(a_i \in L) \wedge \exists(a_j \in P_i) \wedge a_i <_C a_j \wedge a_j \not<_C a_i \wedge \nexists(a_k \in P_j) \wedge a_k <_C a_i \wedge a_i \not<_C a_k \wedge \exists(a_l \in P_n) \wedge a_i <_C a_l \rightarrow \text{deadlock}(\Omega);$$

$$L_1 \subset A(\Omega_1) \wedge L_2 \subset A(\Omega_2) \wedge a_i \in L_1 \cap L_2 \wedge \exists(a_j \in A(\Omega_1)) \wedge a_i <_C a_j \wedge (\nexists(a_k \in A(\Omega_1)) \wedge a_k <_C a_i \vee \text{deadlock}(\Omega_1)) \wedge a_l \in A(\Omega_2) \wedge a_i <_C a_l \rightarrow \text{deadlock}(\Omega_2).$$

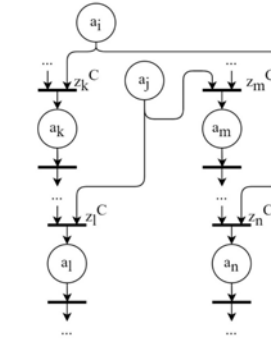


Рис. 2. Пример РСПСС с некорректным порядком взятия блокировок.

Fig. 2. An example of ExPNSR with an incorrect order of taking locks.

Поиск взаимных блокировок, согласно описанным правилам, выполняется следующим образом: во всех параллельных путях отыскиваются позиции взятия блокировки, после чего проверяется наличие операции снятия блокировки в этих же путях. Если путь по управлению не содержит ни одной операции взятия блокировки, он считается свободным от взаимной блокировки. Позиция снятия блокировки определяется как вход перехода по управлению, выходом которого является позиция блокировки.

В модели РСПСС этим правилам соответствуют схемы, представленные на рис. 3.

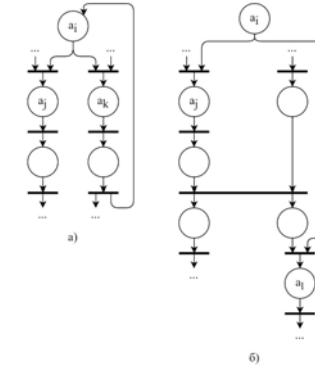


Рис. 3. Пример РСПСС с отсутствием освобождения блокировки (а – в одной параллельной области, б – в двух).

Fig. 3. An example of ExPNSR with no lock release (a – in one parallel region, b – in two).

4.4 Пример

Рассмотрим пример процесса обнаружения взаимоблокировки. Для этого несколько модифицируем ранее рассмотренную программу.

01 var x int	08 // Поток 2
02 var mtx sync.Mutex	09 go func() {
03 // Поток 1	10 y := x
04 go func() {	11 }()
05 mtx.Lock()	12 mtx.Lock()
06 x = 10	13 x = 20
07 }()	14 mtx.Unlock()

Как видно из программы, был добавлен примитив взаимного исключения `mtx` и его блокировка в двух параллельных путях, но в одном из них отсутствует разблокировка. Модель РСПСС для этой программы представлена на рис. 4, где для упрощения восприятия в качестве семантических связей отображены только те из них, что связаны с операциями с взаимным исключением, прочие же остались без изменений. С той же целью позиция `lock`, представляющая собой позицию блокировки, выделена особым образом.

В соответствии с представленным методом выполним анализ полученной модели.

- 1) Идентификация параллельных областей и входящих в них параллельных путей по управлению.

Как было описано ранее, для определения состава параллельных областей пути по управлению формируются, ориентируясь только на позиции, находящиеся в участках, ограниченных переходами типа `Fork`, `Join` и `Synchro`. В представленной модели переходами типа `Fork` являются переходы по управлению 2 и 5:

$$\begin{aligned} P_1 &= (A = \{3,4,5\}, Z^C = \{3,4\}); \\ P_2 &= (A = \{6\}, Z^C = \emptyset); \\ P_3 &= (A = \{7,8,9,10,11,15\}, Z^C = \{6,7,8,9,12\}); \\ P_4 &= (A = \{12,13,14\}, Z^C = \{10,11\}). \end{aligned}$$

Параллельные участки, в соответствии с порождающими их переходами по управлению типа `Fork`, распределяются по параллельным областям следующим образом: $\Omega_1 = \{P_1, P_2, P_3, P_4\}$; $\Omega_2 = \{P_3, P_4\}$.

Следуя правилу определения параллельных участков, определим какие из них соответствуют отношению. В результате получаем $P_1 \parallel_c P_2$, $P_1 \parallel_c P_3$, $P_1 \parallel_c P_4$, $P_3 \parallel_c P_4$. P_2 и P_3 , а также P_2 и P_4 не параллельны, так как для них нарушается условие определения 4: позиция 6 пути P_2 предшествует по управлению всем позициям путей P_3 и P_4 .

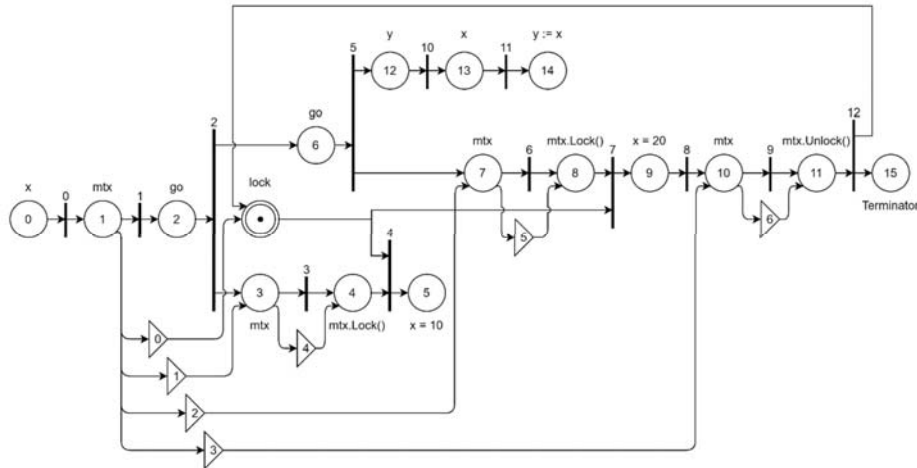


Рис. 4. Пример РСПСС для программы со взаимоблокировкой.
Fig. 4. An example of ExPNRS for the program with deadlock.

- 2) Определение позиций блокировки.

В качестве позиции блокировки обнаруживается позиция `lock`, так как для неё выполняется условие, описанное в определении 6.

- 3) Анализ порядка взятия блокировки.

Так как в программе присутствует только один примитив взаимного исключения, для необходимости проведения данной проверки не выполняется условие наличия в параллельной области как минимум двух позиций блокировки.

- 4) Анализ отсутствия освобождения блокировки.

Так как в параллельной области Ω_1 отсутствует освобождение блокировки позиции `lock`, а в параллельной области Ω_2 имеется повторное взятие блокировки, то можно утверждать о наличии взаимоблокировки в программе.

Таким образом, можно утверждать, что метод позволяет формально, на основе анализа структуры модели РСПСС, выявить оба классических сценария возникновения взаимных блокировок в параллельных программах.

5. Метод поиска гонок данных в параллельных программах

Метод статической верификации для поиска гонок данных основан на анализе полной модели РСПСС, включая как подсеть по управлению (порядок выполнения операций), так и семантическую подсеть (зависимости по данным). Ключевая идея заключается в идентификации операций чтения и записи, обращающихся к одной и той же области памяти (позиции) из параллельных потоков выполнения, без надлежащей синхронизации.

Процесс анализа делится на два основных этапа, зависящих от структуры сети:

- 1) обнаружение гонок данных (data race) вне критических секций;
- 2) обнаружение состояний гонки (race conditions) внутри и между критическими секциями, когда результат зависит от порядка выполнения синхронизированных участков кода.

Основой для анализа является идентификация в модели параллельных путей выполнения, разделение позиций на классы чтения и записи и определение позиций блокировки.

Таким образом, можно выделить основные шаги, необходимые для поиска гонок данных:

- 1) идентификация параллельных областей и входящих в них параллельных путей по управлению;
- 2) определение позиций блокировки;
- 3) идентификация типов операций;
- 4) обнаружение гонок данных вне критических секций;
- 5) обнаружение состояний гонки с учетом критических секций.

Формирование структуры параллельных областей и позиций блокировки выполняется аналогично шагу, описанному для анализа взаимоблокировок. Рассмотрим подробнее правила и определения, связанные с критическими секциями и гонками данных.

5.1 Идентификация типов операций

Гонки данных в параллельных программах могут быть представлены в трех формах:

- «чтение-чтение» – одновременное выполнение операций чтения одной области памяти;
- «чтение-запись» – одновременное выполнение операций чтения и записи в одной области памяти;
- «запись-запись» – одновременное выполнение операций записи в одной области памяти.

Однако гонка данных «чтение-чтение» не представляет интереса, так как не несёт угрозы выполнению программы, именно поэтому в рамках работы выполняется обнаружение оставшихся двух типов.

Прежде чем искать конфликты, необходимо идентифицировать тип операции с памятью: чтение или запись.

Позиция записи представляет собой операцию, которая изменяет значение в некоторой области памяти. Позиция a_w считается операцией записи в память a_i , если:

- является выходом семантического перехода, входом которого является a_i (память используется для записи);
- является входом другого семантического перехода, выходом которого является a_i (новое значение записывается в память);
- a_i предшествует по управлению позиции записи a_w , что гарантирует, что память выделена до операции записи.

Формально это определяется следующим правилом:

$$\exists(a_i \in A) \wedge \exists(a_w \in A) \wedge \exists(z_i^S \in Z^S) \wedge \exists(z_j^S \in Z^S) \wedge a_w \in O_A(z_i^S) \wedge a_w \in I_A(z_j^S) \wedge a_i \in O_A(z_j^S) \wedge a_i <_c a_w \rightarrow \text{write}(a_w, a_i).$$

Позиция чтения представляет собой операцию, которая использует значение из области памяти, но не изменяет его. Позиция a_r считается операцией чтения из памяти a_i , если:

- является выходом семантического перехода, входом которого является a_i (значение считывается из памяти);
- не существует семантического перехода, где a_r была бы входом, а a_i – выходом (то есть эта операция не перезаписывает память).

Формально это определяется следующим правилом:

$$\exists(a_i \in A) \wedge \exists(a_r \in A) \wedge \exists(z_i^S \in Z^S) \wedge \nexists(z_j^S \in Z^S) \wedge a_r \in O_A(z_i^S) \wedge a_i \in I_A(z_i^S) \wedge a_i \in O_A(z_j^S) \wedge a_r \in I_A(z_j^S) \rightarrow \text{read}(a_r, a_i).$$

Классификация позиций, согласно приведённым правилам, выполняется путём обхода всех позиций сети и анализа связанных с ними семантических переходов. Позиции, которые невозможно отнести к чтению или записи (например, выделение памяти, вызов функции и т.п.), исключаются из дальнейшего анализа. Помимо самой классификации, на этом этапе также фиксируется позиция выделения памяти, с которой связаны операции чтения или записи.

5.2 Обнаружение гонок данных вне критических секций

В соответствии с этим гонку «чтение-запись» можно определить как существование в параллельных путях P_i и P_j операций, обращающихся к одной и той же области памяти a_i , при этом в одном пути находится операция записи a_w , а в другом – чтения a_r . Формально это описывается правилом:

$$\exists(a_i \in A) \wedge \exists(a_w \in P_i) \wedge \exists(a_r \in P_j) \wedge P_i ||_c P_j \wedge \text{read}(a_r, a_i) \wedge \text{write}(a_w, a_i) \rightarrow DR_{rw}(a_r, a_w).$$

Гонка «запись-запись» в свою очередь возникает, когда в параллельных путях P_i и P_j существуют операции записи в одну и ту же область памяти a_i . Наличие гонки «запись-запись» формализуется аналогично:

$$\exists(a_i \in A) \wedge \exists(a_{w,i} \in P_i) \wedge \exists(a_{w,j} \in P_j) \wedge P_i ||_c P_j \wedge \text{write}(a_{w,i}, a_i) \wedge \text{write}(a_{w,j}, a_i) \rightarrow DR_{ww}(a_{w,i}, a_{w,j}).$$

Для обнаружения гонки данных, согласно данным правилам, выполняется попарная проверка позиций, классифицированных как чтение или запись, связанных с одной и той же позицией

выделения памяти и находящихся в двух параллельных путях по управлению. Если путь не содержит операций чтения или записи, он признаётся свободным от гонки данных.

5.3 Обнаружение состояний гонки с учетом критических секций

Наличие в параллельной программе синхронизаций, предотвращает одновременное выполнение операций внутри критической секции, поэтому гонки данных внутри одной секции невозможны. Однако может возникнуть состояние гонки – когда результат программы зависит от порядка, в котором параллельные потоки попадают в свои критические секции.

Критическая секция представляет собой множество позиций, заключённых между захватом (переход по управлению, где позиция блокировки a_m является входом) и освобождением (переход по управлению, где a_m является выходом) одной и той же блокировки в рамках одного пути. Формально это описывается следующим правилом:

$$\begin{aligned} & \exists(a_m \in A) \wedge (\text{mutex}(a_m) \vee \text{semaphore}(a_m)) \wedge \exists(z_i^C \in Z^C) \wedge \exists(z_j^C \in Z^C) \wedge \\ & a_m \in I_A(z_i^C) \wedge a_m \in I_A(z_j^C) \wedge \exists(z_k^C \in Z^C) \wedge \exists(z_l^C \in Z^C) \wedge a_m \in O_A(z_k^C) \wedge a_m \in O_A(z_l^C) \rightarrow \\ & \forall \left(\begin{aligned} & a_i \in P_i \wedge a_j \in P_j \wedge P_i ||_c P_j \wedge \left(\begin{aligned} & a_i \in O_A(z_i^C) \vee a_i \in I_A(z_k^C) \\ & \vee (O_A(z_i^C) <_c a_i \wedge a_i <_c I_A(z_k^C)) \end{aligned} \right) \wedge \\ & \left(\begin{aligned} & a_j \in O_A(z_j^C) \vee a_j \in I_A(z_l^C) \\ & \vee (O_A(z_j^C) <_c a_j \wedge a_j <_c I_A(z_l^C)) \end{aligned} \right) \end{aligned} \right) \wedge a_i \in C_{m,i} \wedge a_j \in C_{m,j}, \end{aligned}$$

где $C_{m,i}$, $C_{m,j}$ – множества позиций, формирующих критические секции, образованные позицией блокировки a_m в параллельных путях P_i и P_j соответственно.

Одним из эффектов того, что операции в параллельных критических секциях не могут выполняться одновременно, является независимость операций записи. То есть они не конкурируют за память в процессе своего выполнения, и, следовательно, некорректного результата не возникает. В связи с этим при анализе состояния гонки между критическими секциями следует уделять внимание только ситуации, аналогичной гонке данных «чтение-запись». Оно возникает при следующих условиях:

- 1) в одной критической секции ($C_{m,i}$) есть операция чтения ($a_{r,i}$), перед которой в этой же секции нет операции записи в ту же память a_i ;
- 2) в другой параллельной критической секции ($C_{m,j}$) есть операция записи ($a_{w,j}$), и она является последней операцией записи в этой секции (после нее в секции нет других операций записи в a_i).

Формально это описывается правилом:

$$\begin{aligned} & \exists(a_{r,i} \in C_{m,i}) \wedge \exists(a_i \in A) \wedge \text{read}(a_{r,i}, a_i) \wedge \nexists(a_{w,i} \in C_{m,i}) \wedge \text{write}(a_{w,i}, a_i) \wedge \\ & a_{w,i} <_c a_{r,i} \wedge \exists(a_{w,j} \in C_{m,j}) \wedge \text{write}(a_{w,j}, a_i) \wedge \nexists(a_{k,j} \in C_{m,j}) \wedge \\ & \text{write}(a_{k,j}, a_i) \wedge a_{w,j} <_c a_{k,j} \rightarrow DR_{rw}(a_{r,i}, a_{w,j}). \end{aligned}$$

Согласно данному правилу, обнаружение состояния гонки выполняется путём попарной проверки позиций чтения и записи, относящихся к одной и той же позиции выделения памяти и находящихся в одной критической секции двух параллельных путей по управлению. При этом в качестве позиции записи учитывается только последняя операция записи в данную область памяти внутри критической секции; для этого все подходящие позиции записи проверяются на отношение предшествования по управлению. Если путь не содержит операций чтения или записи либо не имеет критической секции, он считается свободным от состояния гонки.

5.4 Пример

В качестве примера рассмотрим программу, РСРСС которой представлена на рис. 1. В соответствии с представленным методом выполним следующие шаги.

- 1) Идентификация параллельных областей и входящих в них параллельных путей по управлению.

Как было описано ранее, для упрощения анализа сформируем пути по управлению, ориентируясь только на позиции, находящиеся в участках, формируемых переходами типа Fork (1 и 2):

$$\begin{aligned} P_1 &= (A = \{2\}, Z^C = \emptyset); \\ P_2 &= (A = \{3\}, Z^C = \emptyset); \\ P_3 &= (A = \{4, 8\}, Z^C = \{5\}); \\ P_4 &= (A = \{5, 6, 7\}, Z^C = \{3, 4\}). \end{aligned}$$

Параллельные участки, в соответствии с порождающими их переходами по управлению типа Fork, распределяются по параллельным областям следующим образом

$$\Omega_1 = \{P_1, P_2, P_3, P_4\}; \Omega_2 = \{P_3, P_4\}.$$

Следуя правилу определения параллельных участков, определим какие из них соответствуют отношению. В результате получаем, что $P_1 \parallel_c P_2, P_1 \parallel_c P_3, P_1 \parallel_c P_4, P_3 \parallel_c P_4$. В то же время P_2 и P_3 , а также P_2 и P_4 не параллельны, так как для них нарушается условие определения 4: позиция 3 пути P_2 предшествует по управлению всем позициям путей P_3 и P_4 .

- 2) Определение позиций блокировки.

В модели программы отсутствуют позиции блокировки.

- 3) Идентификация типов операций.

Анализируя позиции в полученных параллельных участках и связанные с ними переходы по семантическим связям, получаем следующее разбиение операций по классам: write(2,0); write(4,0); write(7,5); read(6,0); read(7,6).

Можно заметить, что позиция 7 помечается одновременно и как позиция чтения памяти в позиции 6, и как позиция записи в область памяти позиции 5. Это связано с тем, что для записи в память переменной y используется ранее считанное значение переменной x . На практике подобное чтение (read(7,6)) не будет приводить к гонке данных, так как в каждой операции переменные считываются в индивидуальные области, что можно наблюдать, например, на рис. 4, где отдельные позиции 3, 7 и 10 описывают считанное значение примитива взаимного исключения.

- 4) Обнаружение гонок данных вне критических секций.

Анализируя полученные параллельные участки и классы операций, получаем следующие отношения гонок данных:

- $DR_{ww}(2,4)$ – позиции 2 и 4 представляют собой запись в одну и ту же область памяти позиции 0 и принадлежат параллельным участкам P_1 и P_3 соответственно;
- $DR_{rw}(6,2)$ – позиции 6 и 2 представляют собой чтение и запись одной и той же области памяти позиции 0 и принадлежат параллельным участкам P_4 и P_1 соответственно;
- $DR_{rw}(6,4)$ – позиции 6 и 4 представляют собой чтение и запись одной и той же области памяти позиции 0 и принадлежат параллельным участкам P_4 и P_3 соответственно.

- 5) Обнаружение состояний гонки с учетом критических секций

В связи с отсутствием в модели позиций блокировки, данный этап анализа не выполняется.

Таким образом, метод позволяет формально выявить как прямые гонки данных из-за отсутствия синхронизации, так и более сложные недетерминированные поведения (состояния гонки) в синхронизированном коде, анализируя взаимодействие потоков управления и потоков данных в модели РСРСС.

6. Экспериментальная проверка работоспособности метода

В соответствии с представленным методом был разработан прототип программного средства статической верификации программного обеспечения SPNAn (Semantic Petri Net Analyzer). Прототип реализован для анализа программ на языке Go и способен обрабатывать линейные программы без ветвлений и циклов. В качестве инструмента синхронизации учитывается только примитив взаимного исключения (sync.Mutex). Система состоит из следующих компонентов:

- модуль построения РСРСС – принимает на вход абстрактное синтаксическое дерево исходного кода, сгенерированное стандартными средствами языка Go, и преобразует его в формальную модель РСРСС, устанавливая полное соответствие между операциями программы и позициями сети, а также их информационные зависимости и зависимости по управлению;
- модуль верификации – выполняет основные аналитические действия с полученной моделью РСРСС: определяет параллельные области и их состав, определяет тип операций и позиции блокировки, выполняет проверку модели на наличие гонок данных и взаимных блокировок;
- система отчетов – предоставляет пользователю информацию об обнаруженных ошибках по результату верификации.

SPNAn реализован в формате консольного приложения, что также обеспечивает его возможность применения в процессах непрерывной интеграции (CI/CD). Потенциальным недостатком реализованного решения является повышенная чувствительность к комбинаторному взрыву числа состояний программы при построении модели.

Исследование разработанного средства выполнялось на материалах набора данных GoBench [31, 32]. Данный набор содержит более 100 параллельных программ на языке Go, содержащих различные ошибки. Исходя из возможностей реализованной системы из набора были отобраны 38 программ для использования в качестве тестов: 20 тестов, содержащих гонки данных (наборы grps, etcd, istio, serving, kubernetes), и 18 тестов, содержащих различные взаимоблокировки (наборы hugo, grps, etcd, cockroach, synching, moby, kubernetes).

Для языка Go в литературе предлагается ряд инструментов анализа, направленных на обнаружение ошибок параллелизма. GoPV [25] и GoGuard [26] ориентированы на блокирующие ошибки при работе с примитивами взаимного исключения. GoDetector [27] и GCatch [28] специализируются на дефектах каналов и WaitGroup. Godel2 [29-30] представляет собой верификатор моделей (прототип) для проверки свойств живости и безопасности. Все перечисленные инструменты имеют узкую направленность на отдельные классы примитивов и не обеспечивают унифицированного анализа как гонок данных, так и взаимных блокировок в рамках единой модели. Прямое сравнение SPNAn с каждым из них потребует адаптации их к тестовым наборам и формирования единых критериев, что не всегда возможно из-за пересекающихся классов проверок (например, SPNAn в настоящее время не работает с каналами и WaitGroup, а GCatch с примитивами взаимного исключения) или ограниченности прототипов. Поэтому для демонстрации работоспособности предлагаемого метода на текущем этапе выбран динамический анализатор ThreadSanitizer –

стандарт для обнаружения гонок данных в Go. Эффективность SPNAn в выявлении взаимных блокировок подтверждена отдельно на тестах GoBench, что в совокупности доказывает практическую ценность подхода.

Результаты сравнительного тестирования обоих инструментов на наборе тестов GoBench представлены в табл. 1 и табл. 2.

Табл. 1. Результаты верификации агрегированных тестовых наборов GoBench на гонки данных.
Table 1. Verification results of aggregated GoBench test suites for data races.

	gRPC	etcd	istio	serving	kubernetes
ThreadSanitizer	2/2	3/3	3/3	2/2	10/10
SPNAn	2/2	2/3	3/3	2/2	10/10

Табл. 2. Результаты верификации агрегированных тестовых наборов GoBench на взаимные блокировки.
Table 2. Verification results of aggregated GoBench test suites for deadlocks.

	hugo	gRPC	etcd	cockroach	syncthing	moby	kubernetes
ThreadSanitizer	0/2	0/2	0/3	0/4	0/1	0/4	0/2
SPNAn	1/2	2/2	3/3	2/4	0/1	3/4	2/2

В результате экспериментов гонки данных были обнаружены при помощи SPNAn в 19 тестах (кроме etcd_8194), взаимные блокировки в 12 тестах (кроме hugo_5379, etcd_6708, cockroach_584, cockroach_10214, syncthing_4829, moby_7559). Неполное обнаружение как гонок данных, так и взаимоблокировок может быть объяснено несовершенством прототипа системы в части построения модели: наличием циклов в программе и использованием в качестве примитива синхронизации RWMutex. Указанные недостатки могут быть устранены при последующем развитии системы путем добавления в алгоритм построения модели учёта указанных элементов.

В сравнении с ThreadSanitizer система SPNAn показала следующие результаты – в отличие от стандартного средства, она способна выполнять обнаружение взаимоблокировок, а обнаружение гонок данных выполняется в случае, если модуль построения модели программы способен корректно преобразовать структуру программы. Также следует отметить, что система SPNAn не сгенерировала ложных срабатываний в случае корректного построения модели, но данный фактор может быть обусловлен простотой анализируемых тестов, и отсутствие ложноположительных срабатываний в более сложных системах не гарантируется.

7. Дальнейшая работа

В качестве возможного дальнейшего развития исследования можно выделить несколько направлений.

- Развитие процесса построения модели: расширение охвата языковых конструкций (анализ циклов, ветвлений, учет сторонних библиотек и т.д.). Это позволит значительно повысить точность и полноту анализа.
- Оптимизация модели - исследование методов упрощения РСПСС (редукции) для борьбы с комбинаторным взрывом числа состояний программы без потери точности анализа. Это позволит применять метод к реальным, крупным программным проектам. Вместо анализа всей программы целиком можно будет использовать техники абстракции и редукции для фокусировки на наиболее критичных или подозрительных участках кода. Это напрямую повлияет на масштабируемость инструмента и его внедрение в промышленную разработку.
- Оптимизация процесса верификации, например, путем внедрения распараллеливания, разработки более эффективных алгоритмов обхода состояния

сети Петри для сокращения времени анализа больших моделей. Это сделает инструмент практичным для повседневного использования разработчиками. Внедрение распараллеливания позволит существенно сократить временные затраты на процесс верификации, а разработка более эффективных алгоритмов обхода позволит не перебирать все состояния слепо, а целенаправленно проверять только те сценарии, которые потенциально ведут к ошибкам.

- Расширение класса обнаруживаемых ошибок, что позволит развить инструмент из узкоспециализированного верификатора гонок и взаимоблокировок в универсальную платформу для диагностики параллелизма.
- Гибридизация путем интеграции с другими методами для компенсации узких мест статической верификации.
- Адаптация метода для других парадигм параллельного программирования и языков программирования, что позволит получить универсальное инструментальное решение, способное работать с различными технологиями. Однако это потребует дополнительной работы по созданию инструмента преобразования исходного кода в формат, удобный для анализа.

На текущем этапе достигнуты следующие ключевые результаты:

- предложен метод статической верификации параллельной программы на наличие гонок данных, состояния гонок и взаимоблокировок, отличающийся от известных представлением её в виде РСПСС с обеспечением возможности автоматического установления соответствия между операторами программы и элементами РСПСС, включая элементы синхронизации;
- разработан функционирующий прототип системы SPNAn для языка Go, реализующей описанный метод;
- экспериментально подтверждена эффективность метода для обнаружения гонок данных и взаимных блокировок.

Достигнутые результаты демонстрируют практическую состоятельность предложенного подхода и создают основу для его дальнейшего развития в указанных направлениях.

8. Заключение

В работе предложен подход к статической верификации. Предложена математическая модель параллельной программы в виде РСПСС и получен набор правил, формализующих идентификацию критических секций, взаимоблокировок и гонок данных, позволяющих обнаруживать критические ошибки параллельного выполнения.

Создан прототип системы статической верификации SPNAn (Semantic Petri Net Analyzer), реализующий автоматическое построение РСПСС из исходного кода на языке Go и последующий анализ модели, подтверждающий практическую применимость предложенного подхода.

Обоснована эффективность выбранного подхода в сравнении с существующими методами верификации. Показано, что статическая верификация на основе проверки модели (Model Checking) с использованием формализма РСПСС позволяет совместить приемлемую полноту проверяемых свойств с возможностью автоматизации и практической реализуемостью инструментальных средств.

Научная и практическая значимость работы заключается в создании теоретического фундамента и практического инструментария для повышения надежности параллельного программного обеспечения. Разработанные методы могут быть применены при создании критических систем, где недопустимы ошибки синхронизации и гонки данных.

Таким образом, работа вносит существенный вклад в область формальной верификации параллельных программ и создает основу для разработки новых эффективных средств обеспечения корректности программного обеспечения в условиях массового распространения многопроцессорных и распределенных вычислительных систем.

Список литературы / References

- [1]. Гурий Р.Е., Рудаков И.В., Ребриков А.В. Методы верификации программного обеспечения. Машиностроение и компьютерные технологии, 2015, вып. 10, стр. 235-250 / Gurin R.E., Rudakov I.V., Rebrikov A.V. Methods of software verification. Science and Education of the Bauman MSTU, 2015, no. 10, pp. 235-250 (in Russian).
- [2]. Helmbold D., McDowell C. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, 1994.
- [3]. Трифанов В.Ю., Цителов Д.И. Динамические средства обнаружения гонок в параллельных программах. КИО, 2011, вып. 6, стр. 3-15 / Trifanov V.Yu., Tsitelov D.I. Dynamic tools for detecting races in parallel programs. KIO, 2011, no. 6, pp. 3-15 (in Russian).
- [4]. Мордань В.О. Комбинация методов статической верификации композиции требований. Труды ИСП РАН, 2017, том 29, вып. 3, стр. 151-170. / Mordan V.O. Combination of static verification methods for checking requirements composition. Trudy ISP RAN/Proc. ISP RAS, 2017, vol. 29, issue 3, pp. 151-170 (in Russian).
- [5]. Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. Под ред. А.П. Ершова. М.: Радио и связь, 1988, 256 с. / Nepomnyashchiy V.A., Ryakin O.M. Applied Methods for Programs Verification. Moscow: Radio i svyaz', 1988, 256 p. (in Russian).
- [6]. Rondon P. Liquid Types: Thesis for the Degree of Doctor of Philosophy. SanDiego: University of California, 2012, 265 p.
- [7]. Кулямин В.В. Методы верификации программного обеспечения. М.: Институт системного программирования Российской академии наук, 2008, 111 с. / Kulyamin V.V. Methods of software verification. Moscow: Institute for System Programming of the Russian Academy of Sciences, 2008, 111 p. (in Russian).
- [8]. Lamport L. The temporal logic of actions. ACM Trans. Program. Lang. Syst., 1994, vol. 16, no. 3, pp. 872-923.
- [9]. Mauro Pezzè, Richard N. Taylor, Michal Young. Graph models for reachability analysis of concurrent programs. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 4, no. 2, pp. 171-213.
- [10]. Тихомиров В.А., Тимофеев С.Г., Мошкова Е.А. Модифицированный метод верификации программ на модели. Вестник Российской академии естественных наук, 2018, вып. 3, стр. 118-121. / Tikhomirov V.A., Timofeev S.G., Moshkova E.A. Modified model checking verification method. Bulletin of the Russian Academy of Natural Sciences, 2018, no. 3, pp. 118-121. (in Russian).
- [11]. Petri C. A. Kommunikation mit Automaten: Dissertation. 1962.
- [12]. Keller R. M. Formal Verification of Parallel Programs. Communications of the ACM, 1976, vol. 19, no. 7, pp. 371-384.
- [13]. He X. A comprehensive survey of Petri net modeling in software engineering. International Journal of Software Engineering and Knowledge Engineering, 2013, vol. 23, no. 5, pp. 589-625.
- [14]. Kavi K.M., Moshtaghi A., Chen D. Modeling multithreaded applications using Petri nets. International Journal of Parallel Programming, 2002, vol. 30, pp. 353-371.
- [15]. Barkaoui K., Couvreur J. M., Klai K. On the equivalence between liveness and deadlock-freeness in Petri nets. In: International Conference on Application and Theory of Petri Nets. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 90-107.
- [16]. Guo X., Wang S., You D., Li Z., Jiang X. A Siphon-Based Deadlock Prevention Strategy for S3PR. IEEE Access, 2019, vol. 7, pp. 86863-86873.
- [17]. Xiang D., Liu G., Yan C., Jiang C. Detecting data-flow errors based on Petri nets with data operations. IEEE/CAA Journal of Automatica Sinica, 2018, vol. 5, no. 1, pp. 251-260.
- [18]. Liao H., Wang Y., Cho H.K., Stanley J., Kelly T., Lafortune S., Mahlke S., Reveliotis S. Concurrency bugs in multithreaded software: modeling and analysis using Petri nets. Discrete Event Dynamic Systems, 2013, vol. 23, pp. 157-195.
- [19]. Wang Y., Liao H., Reveliotis S., Kelly T., Mahlke S., Lafortune S. Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In: Proceedings of the 48th IEEE

- Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference, 2009, pp. 4973-4978.
- [20]. Liao H., Wang Y., Stanley J., Lafortune S., Reveliotis S., Kelly T., Mahlke S. Eliminating concurrency bugs in multithreaded software: a new approach based on discrete-event control. IEEE Transactions on Control Systems Technology, 2013, vol. 21, no. 6, pp. 2067-2082.
 - [21]. Sun J., Liu G., Xiang D., Jiang C. A Petri-net-based method for detecting bugs in multiple threads. In: 2019 IEEE 16th International Conference on Networking, Sensing and Control (ICNSC), Banff, AB, Canada, 2019, pp. 150-156.
 - [22]. Kharitonov D.I., Odyakova D.S. Modelling race conditions in multithreading programs in terms of Petri nets. IOP Conference Series: Materials Science and Engineering, 2020, vol. 734, no. 1.
 - [23]. Voloshko A., Kryukov O. Extended Petri Nets Based Approach for Simulation of Distributed Manufacturing Processes. In: The 9th Mediterranean Conference on Embedded Computing (MECO 2020), 2020, pp. 508-511.
 - [24]. Крюков О.С., Волошко А.Г., Ивутин А.Н. Моделирование примитивов синхронизации параллельных программ // Моделирование и анализ информационных систем, 2025, Т. 32, вып. 2, стр. 150-171. / Kryukov O.S., Voloshko A.G., Ivutin A.N. Modeling of Parallel Program Synchronization Primitives. Modeling and Analysis of Information Systems, 2025, vol. 32, no. 2, pp. 150-171 (in Russian).
 - [25]. Xu X., Song W., Huang J. GoPV: A Static Analyzer for Finding Blocking Concurrency Bugs Due to Misuse of Shared-Memory Synchronization in Go. In: Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25). New York: ACM, 2025, pp. 36-40.
 - [26]. Liu B., Joshi D. GoGuard: Efficient Static Blocking Bug Detection for Go. In: Giacobazzi R., Gorla A. (eds.) Static Analysis. SAS 2024. Lecture Notes in Computer Science, vol. 14995. Springer, 2025.
 - [27]. Zhang D., Qi P., Zhang Y. GoDetector: Detecting Concurrent Bug in Go. IEEE Access, 2021, vol. 9, pp. 136302-136312.
 - [28]. Liu Z., Zhu S., Qin B., Chen H., Song L. Automatically detecting and fixing concurrency bugs in go software systems. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21). New York: ACM, 2021, pp. 616-629.
 - [29]. Lange J., Ng N., Toninho B., Yoshida N. A static verification framework for message passing in go using behavioural types. In: Proceedings of the 40th International Conference on Software Engineering (ICSE '18). New York: ACM, 2018, pp. 1137-1148.
 - [30]. Gabet J., Yoshida N., Pape T., Hirschfeld R. Static Race Detection and Mutex Safety and Liveness for Go Programs. In: 34th European Conference on Object-Oriented Programming (ECOOP 2020), 2020.
 - [31]. Yuan T., Li G., Lu J., Liu C., Li L., Xue J. GoBench: a benchmark suite of real-world go concurrency bugs. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21). IEEE Press, 2021, pp. 187-199.
 - [32]. gobench-cgo21.tar. Available at: https://figshare.com/articles/software/gobench-cgo21_tar/13257023?file=26178281, accessed: 04.11.2025.

Информация об авторах / Information about authors

Олег Сергеевич КРЮКОВ – старший преподаватель кафедры Вычислительная техника Института прикладной информатики и компьютерных наук Тульского государственного университета. Сфера научных интересов: параллельное программирование, верификация программного обеспечения, построение и анализ моделей программ.

Oleg Sergeevich KRYUKOV – Senior Lecturer at the Department of Computer Engineering, Institute of Applied Informatics and Computer Science, Tula State University. Research interests: parallel programming, software verification, software model construction and analysis.

Анна Геннадьевна ВОЛОШКО – кандидат технических наук, доцент кафедры Вычислительная техника Института прикладной информатики и компьютерных наук Тульского государственного университета. Сфера научных интересов: сети Петри, гетерогенные вычислительные системы, параллельное программирование, оптимизация, моделирование и симуляция.

Anna Gennadievna VOLOSHKO – Cand. Sci. (Tech.), Associate Professor at the Department of Computer Technology, Institute of Applied Informatics and Computer Science, Tula State University. Research interests: Petri nets, heterogeneous computing systems, parallel programming, optimization, modeling and simulation.

Алексей Николаевич Ивутин – доктор технических наук, заведующий кафедрой Вычислительная техника Института прикладной информатики и компьютерных наук Тульского государственного университета. Сфера научных интересов: гетерогенные вычислительные системы, моделирование и верификация программных систем, управление распределенными программными системами.

Alexey Nikolaevich IVUTIN – Dr. Sci. (Tech.), Head of the Department of Computer Technology, Institute of Applied Informatics and Computer Science, Tula State University. Research interests: heterogeneous computing systems, modeling and verification of software systems, and the management of distributed software systems.