MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 32, NO. 2, 2025
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Modeling of Parallel Program Synchronization Primitives

O. S. Kryukov[1], A. G. Voloshko[1], A. N. Ivutin[1]

DOI: 10.18255/1818-1015-2025-2-150-171

[1]Tula State Uiversity, Tula, Russia

MSC2020: 68Q85
Research article
Full text in Russian

Received September 29, 2024
Revised April 1, 2025
Accepted May 7, 2025

This article is devoted to the problem of verifying parallel programs that may contain special types of errors associated with the synchronization of parallel executed threads and access to shared memory. Such errors include deadlocks and data races. There is a division of parallel program verification methods into static and dynamic. The second ones require running the code and allow to check only the current implementation of the program for races, which, if there are a large number of branches, can lead to missing races. Among static methods, analytical methods (for example, based on deductive analysis) and model checking methods are most widely used. However, they are difficult to implement, and model checking still require a significant amount of manual work from the programmer to build such a model. In this regard, it is necessary to use models that can be built automatically. Previously, the authors developed a model based on an extension of Petri nets, which allows automatic creation based on sequential code and its conversion into parallel code. Automatic creation of a model of a parallel program introduces new, previously unused requirements related to the interaction of parallel threads. Thus, this article discusses the features of modeling using extended Petri nets with semantic relations of the main synchronization primitives implemented in most languages and parallel programming technologies for shared memory systems. In the future, these models will be used to search for data races and deadlocks in parallel programs.

Keywords: verification; parallel program; Petri nets; synchronization primitives

INFORMATION ABOUT THE AUTHORS

Kryukov, Oleg S. (corresponding author)	ORCID iD: 0000-0003-3155-6266. E-mail: ol_kryukov97@mail.ru Postgraduate Student
Voloshko, Anna G.	ORCID iD: 0000-0002-4304-2513. E-mail: atroshina@mail.ru Ph.D, Docent, Associate Professor of the Department of Computer Engineering
Ivutin, Alexey N.	ORCID iD: 0000-0003-2970-2148. E-mail: alexey.ivutin@gmail.com Dr. Sc., Professor, Head of the Department of Computer Engineering

For citation: O. S. Kryukov, A. G. Voloshko, and A. N. Ivutin, “Modeling of parallel program synchronization primitives”, *Modeling and Analysis of Information Systems*, vol. 32, no. 2, pp. 150–171, 2025. DOI: 10.18255/1818-1015-2025-2-150-171.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 32, № 2, 2025
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Моделирование примитивов синхронизации параллельных программ

О. С. Крюков[1], А. Г. Волошко[1], А. Н. Ивутин[1]

DOI: 10.18255/1818-1015-2025-2-150-171

[1]Тульский государственный университет, Тула, Россия

УДК 519.876.5
Научная статья
Полный текст на русском языке

Получена 29 сентября 2024 г.
После доработки 1 апреля 2025 г.
Принята к публикации 7 мая 2025 г.

Данная статья посвящена проблеме верификации параллельных программ, которые могут содержать особые виды ошибок, связанных с синхронизацией параллельно исполняемых потоков и доступом к общей памяти. К таким ошибкам относятся тупики и гонки данных. Существует разделение методов верификации параллельных программ на статические и динамические. Последние требуют запуска кода и позволяют проверить на гонки лишь текущую реализацию программы, что при наличии большого числа ветвлений может привести к пропуску гонок. Среди статических методов наибольшее применение нашли аналитические методы (например, на основе дедуктивного анализа) и методы проверки моделей. Однако они сложны в реализации, а последние по-прежнему требуют от программиста значительного объёма ручной работы для построения модели. В этой связи необходимо использование моделей, которые могут быть построены автоматически. Ранее авторами была разработана модель на основе расширения сетей Петри, позволяющая автоматическое построение на основе последовательного кода и преобразование её в параллельный код. Автоматическое построение модели параллельной программы вводит новые, ранее не использовавшиеся требования, связанные со взаимодействием параллельных потоков. Таким образом, в данной статье рассматриваются особенности моделирования с использованием расширенных сетей Петри с семантическими связями основных примитивов синхронизации, реализуемых в большинстве языков и технологий параллельного программирования для систем с общей памятью. В дальнейшем на основе этих моделей будет проводится поиск гонок данных и тупиков для параллельных программ.

Ключевые слова: верификация; параллельная программа; сети Петри; примитивы синхронизации

ИНФОРМАЦИЯ ОБ АВТОРАХ

Крюков, Олег Сергеевич (автор для корреспонденции)	ORCID iD: 0000-0003-3155-6266. E-mail: ol_kryukov97@mail.ru Аспирант
Волошко, Анна Геннадьевна	ORCID iD: 0000-0002-4304-2513. E-mail: atroshina@mail.ru Канд. тех. наук, доцент, доцент кафедры вычислительной техники
Ивутин, Алексей Николаевич	ORCID iD: 0000-0003-2970-2148. E-mail: alexey.ivutin@gmail.com Доктор тех. наук, профессор, заведующий кафедрой вычислительной техники

Для цитирования: O. S. Kryukov, A. G. Voloshko, and A. N. Ivutin, “Modeling of parallel program synchronization primitives”, *Modeling and Analysis of Information Systems*, vol. 32, no. 2, pp. 150–171, 2025. DOI: 10.18255/1818-1015-2025-2-150-171.

Введение

Современный этап развития вычислительных систем характеризуется использованием параллельных архитектур. И если простые задачи все еще пишутся для работы на одном ядре, то для более сложных задач уже желательно использовать все возможные мощности. Одним из классов таких сложных систем являются интеллектуальные системы, которые часто имеют дело с обработкой больших объемов информации, что достаточно хорошо может быть выполнено в параллельном режиме. В последнее время разрабатывается все больше параллельных алгоритмов для различных интеллектуальных задач, например, для задач анализа данных [1], для задач нечёткой логики [2], нейросетей [3] и т. д.

Однако, программирование для параллельных систем представляет значительную сложность в связи с необходимостью координации выполнения потоков и управления общими данными. Для этого применяются различные механизмы синхронизации. Более того, многопоточные программы могут содержать дополнительно ошибки, связанные с совместным использованием ресурсов, поиск которых сопряжен со значительными сложностями и требует применения вычислительно затратных методов и алгоритмов. Краткий обзор методов проверки параллельных программ представлен в разделе 1. На наш взгляд, наиболее перспективным является метод статического анализа на основе проверки модели. Статические методы в отличие от динамических рассматривают весь код программы, а не только один из вариантов её исполнения, когда множество альтернативных путей развития алгоритма не может быть проанализировано. К сожалению, недостатком любых средств статического анализа кода, и в особенности метода проверки моделей (включая поиск гонок данных), является наличие большего по сравнению с динамическими средствами количества ложноположительных срабатываний, когда анализатор указывает на ошибку при её реальном отсутствии [4]. Чаще всего такое происходит при наличии чередований потоков, организованных нетиповыми способами, которые не учтены в анализаторе.

В качестве модели программного кода для статического анализа параллельных программ можно использовать расширенные сети Петри с семантическими связями (РСПСС) [5]. Как было показано ранее [6], с помощью таких сетей можно представлять параллельные алгоритмы и анализировать наличие семантических отношений между параллельными потоками. В дополнение к операциям создания и завершения потоков, которые представлены соответствующими управляющими переходами fork и join, модель обеспечивает переход типа synchro для барьерной синхронизации. Этих примитивов синхронизации было достаточно для решения задач автоматизации распараллеливания с помощью РСПСС. Анализ изначально параллельных программ требует наряду с проверкой каждого последовательного потока решения более сложной проблемы исследования межпотокового взаимодействия. Вопросы представления последовательных участков кода, а также процедур ветвления и завершения потоков в этой модели несложны и аналогичны представленным в [6]. Представление примитивов синхронизации связано с рядом проблем. Одна из них, учёт особенностей различных способов синхронизации доступа к ресурсам, помимо барьеров в реальных программах, и их представление в выбранной модели. И мы постараемся описать основные из них в этой статье.

1. Обзор методов анализа параллельных программ

Для начала всё же более подробно рассмотрим методы анализа параллельных программ. Двумя наиболее сложными проблемами являются тупики и гонки данных. Тупики возникают, когда ни один поток из некоторой группы потоков параллельной программы не может завершить свою работу по причине ожидания освобождения ресурсов другим потоком этой группы [7, 8]. Обнаружение тупиковых ситуаций чаще всего проводится на основе построения путей выполнения программы (трасс) и анализа отношений ожидания [9, 10], что позволяет определить, существует

ли трасса, содержащая тупиковое исполнение. Кроме того, обнаружение тупиковых ситуаций встроено в некоторые системы разработки и применяет механизм тайм-аутов, заключающийся в анализе времени, проведённого потоком в ожидании: если оно превысило определённый лимит на всех потоках, то выдаётся предупреждение о возникновении взаимной блокировки [7]; но данный способ применим только для обнаружения полной взаимоблокировки программы.

Несмотря на то, что проблема обнаружения тупиков является актуальной, но как было сказано, уже разработаны методы их выявления. Более сложной для обнаружения ошибкой в многопоточных программах являются гонки данных. Это связано с тем, что гонки данных зачастую связаны с трудновоспроизводимыми путями выполнения, что в свою очередь также усложняет и поиск тупиковых состояний. Для поиска гонок данных применяются различные средства, которые разделяют на статические и динамические методы обнаружения гонок данных.

Динамические методы используют информацию, полученную в результате конкретного выполнения параллельной программы. Пути выполнения ограничены теми, которые могут возникнуть при данных конкретных входных данных. Это приводит к уменьшению проверяемых на наличие гонок данных вариантов. Динамические методы обнаружения гонок данных обычно классифицируют на методы post-mortem и методы детектирования «на лету».

Первые анализируют отслеживаемую информацию или воспроизводят программу после выполнения. Например, при работе программы может динамически осуществляться сбор информации о ссылках на память, а при завершении работы выполняться их анализ на наличие конфликтов в параллельных участках (RecPlay [11]).

Методы «на лету» могут быть основаны на анализе отношения «произошло до», при котором проверяется, что операции взаимодействия с памятью при нескольких последовательных запусках осуществляются в одном и том же порядке (например, FastTrack [12], Djit+ [13], ThreadSanitizer [14], и т. д.), анализе блокировок, когда проверяется, что два конфликтующих события удерживают одну блокировку, что говорит об отсутствии гонок данных (например, Eraser [15]), или гибридном анализе (например AccuLock [16], SimpleLock [17], и т. д.).

Статические методы используют только исходный код для обнаружения гонок данных. Эти методы должны учитывать все возможные входные данные и все возможные пути выполнения при задании этих входных данных. Этот подход включает дедуктивный анализ и метод проверки моделей.

Дедуктивный анализ предлагает генерировать набор математических доказательств в соответствии с программным кодом и его спецификациями и выполнять эти спецификации с помощью либо вспомогательных средств доказательства (интерактивных средств доказательства теорем), либо автоматических средств доказательства теорем, в том числе SMT-решателей [18]. Истинность представленных доказательств означает, что система соответствует своей спецификации. Однако этот метод имеет значительную сложность, а объём доказательства корректности может значительно превышать объем исследуемой программы [19]. Метод проверки моделей является одним из широко используемых статических методов анализа параллельных программ [20]. Проверка модели — автоматизированный подход, позволяющий проверить, соответствует ли модель программы заданной спецификации (формальному описанию свойств системы). При этом в качестве описания спецификации и самого программного обеспечения могут использоваться как логико-алгебраические модели [21], так и исполняемые модели [22]. Проверка модели может выполняться путём перебора всех возможных состояний программы или путём задания булевых отношений, связывающих начальное и конечное состояния. Отдельно стоит отметить широкое использование графовых моделей для представления и анализа параллельных программ, отличающихся как наглядностью для человека, так и простотой компьютерного анализа. Среди них стоит отметить модель «опера-

ции — операнды» [23], а также различные модификации сетей Петри [24], в особенности, цветные сети Петри, например, [25—27].

Для использования этого метода требуется согласование между исходным кодом параллельной программы и сгенерированной моделью программы для обнаружения гонки данных и взаимоблокировок с использованием метода проверки модели. Применимость модели РСПСС для этого представления обсуждается далее в этой статье.

2. Расширенные сети Петри с семантическими связями

Приведём описание модели, используемой для представления параллельных программ. Расширенная сеть Петри с семантическими связями является иерархической сетью Петри, позиции данной сети являются математическими объектами, моделирующими операторы алгоритма. Структура сети задается следующим множеством:

$$\Pi = \left\{A, \left\{Z^C, \widetilde{R}^C, \hat{R}^C\right\}, \left\{Z^S, \widetilde{R}^S, \hat{R}^S\right\}\right\},$$

где A — конечное множество позиций, представляющих операции программы; Z^C — конечное множество переходов по управлению, представляющих последовательность выполнения программы; Z^S — конечное множество переходов по семантическим связям, представляющих информационные зависимости между переменными; $\widetilde{R}^C$ — матрица смежности, отображающая множество позиций в множество переходов по управлению; $\hat{R}^C$ — матрица смежности, отображающая множество переходов по управлению в множество позиций; $\widetilde{R}^S$ — матрица смежности, отображающая множество позиций в множество переходов по семантическим связям; $\hat{R}^S$ — матрица смежности, отображающая множество переходов по семантическим связям в множество позиций. Кроме того, на сеть для анализа могут быть наложены дополнительные параметры. $M = \left\{h^C(t), \Lambda^C, \Lambda^S\right\}$, определяющие временные (вектор времени исполнения $h^C(t)$) и логические характеристики РСПСС (вектора условий срабатывания соответствующих переходов Λ^C, Λ^S).

Позиции РСПСС являются математическими объектами, моделирующими процесс выполнения операции исполнительным устройством в ЭВМ. Следует отметить, что сеть строится по принципу декомпозиции, и одна позиция РСПСС может сама представлять собой подсеть. В ЭВМ структура сети может быть представлена с использованием матриц смежности в разреженном виде, так как они преимущественно заполнены нулевыми элементами, что снижает требования к компьютерной памяти.

Переход по управлению в РСПСС является математическим объектом, моделирующим процесс передачи управления по выполнению операций с одного процесса на другой. В случае параллельного исполнения алгоритма возможна передача управления сразу нескольким процессам, что выражается в соответствующем срабатывании переходов и появлении соответствующих фишек в нескольких позициях сети.

Переход по семантическим связям в РСПСС является математическим объектом, устанавливающим семантические ограничения на последовательность смены состояний одного или нескольких элементов системы. Наличие перехода по семантическим связям между двумя позициями a_i и a_j указывает на то, что для выполнения операции, представленной в позиции a_j необходимо наличие информации, порождаемой в позиции a_i. Например, a_i — присвоение значения в переменную x, a_j — считывание значения переменной x.

Графически позиции сети представляются кругами, переходы по управлению — утолщёнными линиями, переходы по семантическим связям — треугольниками. Пример РСПСС для последовательного линейного алгоритма приведен на рисунке 1. Элементы представленной модели можно интерпретировать следующим образом: позиции 0 и 1 можно расценивать как память (например, переменные), хранящую некоторые значения, позиции 2 и 3 — чтение значения из памяти

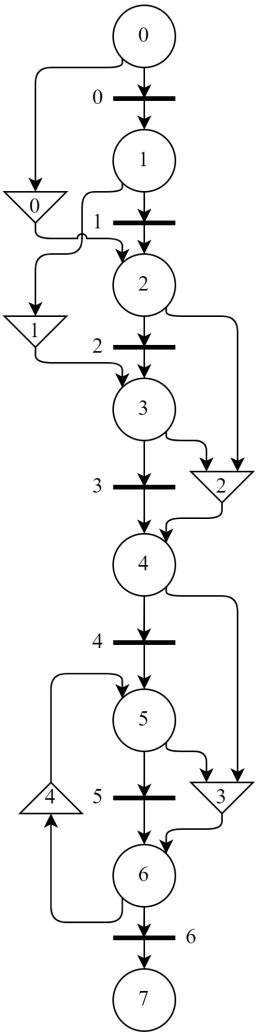

Fig. 1. ExPNSR of sequential linear algorithm

Рис. 1. РСПСС последовательного линейного алгоритма

в позициях 0 и 1 соответственно, позиция 3 — выполнение операции со значениями из 0 и 1, позиция 4 — выполнение некоторой операции со считанными значениями, позиция 5 — некоторая память, отличная от ассоциированной с позициями 0 и 1, позиция 6 — запись результатов операции в позиции 4 в память на позиции 5, позиция 7 — терминатор алгоритма. Таким образом, переход по семантическим связям 0 и 1 показывают чтение конкретной памяти, переход по семантическим связям 2 — использование считанных значений в операции, переход по семантическим связям 3 — использование результата операции для записи в конкретную память, а переход по семантическим связям 4 — зависимость памяти от операции записи. Так как алгоритм линейный, то все позиции последовательно соединены переходами по управлению.

Этому может соответствовать следующая последовательность операций.

```
Начало;
a = 10;
b = 5;
c = a + b;
Конец.
```

Преимуществом такой модели является разделение потока управления и семантических связей. В отличие от классических методов на основе графов операции-операнды, где невозможно четко различить разные ветви условий и параллельные процессы, здесь такое отличие становится очевидным. Кроме того, четко определяются параллельные области, можно варьировать число потоков и анализировать балансировку нагрузки между параллельными потоками. Это позволяет более простыми методами анализировать и проектировать параллельные алгоритмы [28].

Для анализа модели часто необходимо анализировать входные $(I_A(z^S), I_A(z^C))$, описывающие множество позиций, отображаемых в данный переход, и выходные $(O_A(z^S), O_A(z^C))$, описывающие множество позиций, отображаемых из данного перехода, функции переходов.

В РСПСС возможны следующие типы переходов в зависимости от мощности соответствующих множеств $I_A(z)$ и $O_A(z)$: — примитивные переходы, которые имеют одну входную и одну выходную позицию, то есть $|I_A(z_i)| = |O_A(z_i)| = 1$. Так как примитивным может быть любой тип перехода, то его в формуле не специфицируем. Непримитивные переходы по управлению:

- fork, $\left|I_A(z^C)\right| = 1$, $\left|O_A(z^C)\right| > 1$, представляющий начало параллельного вычислительного процесса, когда из одного потока/процесса порождается несколько;
- join, $\left|I_A(z^C)\right| > 1$, $\left|O_A(z^C)\right| = 1$, представляющий завершение параллельного вычислительного процесса, когда несколько параллельных потоков/процессов завершаются и далее продолжает работу только один;
- synchro, $\left|I_A(z^C)\right| = n$, $\left|O_A(z^C)\right| = m$, $n, m > 1$, представляющий синхронизацию, когда несколько потоков/процессов ожидают завершения определённых операций друг от друга для продолжения своей работы. В общем случае какие-то потоки или процессы могут полностью завершить своё выполнение, при этом $m \neq n$.

Для переходов по семантическим связям возможен только один непримитивный переход s-join, $\left|I_A(z^S)\right| > 1$, $\left|O_A(z^S)\right| = 1$, представляющий объединение информации от разных операций для выполнения следующей.

Пример РСПСС для параллельного алгоритма приведён на рисунке 2. По аналогии с РСПСС последовательного алгоритма, здесь также позицию 0 можно расценивать как память, хранящую некоторое значение, позиция 2 — некоторая операция, для вычисления которой не используются сторонние значения, позиции 3 и 5 — операции записи значений в память в позиции 0 (позиция 5 в таком виде может означать запись константы, так как не зависит от какой-либо операции вычисления значений), позиция 8 — чтение значения из памяти в позиции 0, позиция 9 — память отличная от ассоциированной с позицией 0, позиция 10 — запись в память на позиции 9 считанного в позиции 8 значения из позиции 0, позиции 4 и 6 — терминаторы функций соответствующих потоков, позиция 9 — терминатор основной функции. Так как алгоритм параллельный то в нём присутствуют специальные позиции: позиция 1 — порождение параллельного участка, и позиция 7 — объединение параллельных путей алгоритма в один. Помимо примитивных переходов по управлению 0, 2–4, 6–9 в модели присутствуют переход типа Fork под номером 1 и переход типа Join под номером 5.

Этому может соответствовать следующий псевдокод.

```
Начало;
a = 10;
создание потоков 1 и 2;
поток 1:
    Начало;
    a = foo();
    Конец.
поток 2:
    Начало;
```

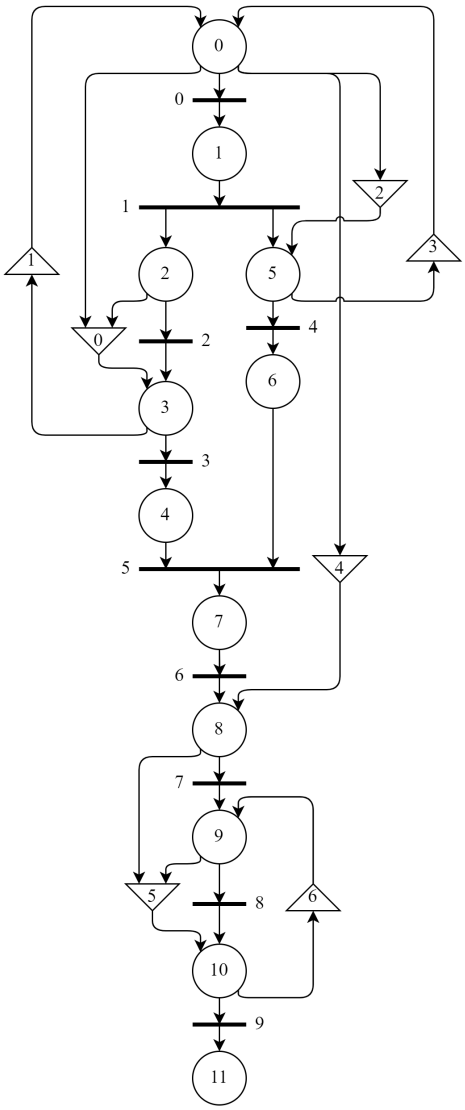

Fig. 2. ExPNSR of concurrent algorithm

Рис. 2. РСПСС для параллельного алгоритма

```
    a = 10;
    Конец.
ожидание завершение потоков 1 и 2;
b = a;
Конец.
```

Каждая позиция РСПСС имеет не более одной дуги, связывающей её 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.

Наличие двух типов переходов приводит к изменению понятия фишки. В классической теории сетей Петри фишка является атомарной структурой. Однако в ряде работ (например, [29]) фишки могут сами представлять собой сложные объекты с сетевой структурой. В РССПСС используется составная фишка T, разделяемой на управляющую фишку (множество фишек) — T^C и семантическую фишку (множество фишек) — T^S. Каждый из данных подтипов фишки формирует поведение соот-

ветствующего типа переходов, объединяясь в позициях в единую фишку $T = T^S + T^C$. Рассмотрим поведение каждого типа фишек в процессе функционирования сети.

Прежде всего следует отметить, что при моделировании вычислений на ЭВМ в любой позиции сети в любой момент времени не может быть более одной объединённой фишки. Наличие иной ситуации свидетельствовало бы, что в одно и то же время на одном исполнительном устройстве (процессоре, ядре) проводится несколько одинаковых вычислений, что не имеет смысла. В системе может быть более одной позиции с фишкой только в том случае, если эти позиции принадлежат разным параллельным путям, т. е. их пути по управлению от начальной позиции сети имеют больше общих переходов типа fork, чем общих переходов типа join.

Для срабатывания перехода по управлению необходимо, чтобы во всех входных позициях перехода было по одной объединенной фишке, если иное не установлено в векторе условий срабатывания перехода Λ^C. При срабатывании управляющая фишка извлекается из соответствующих позиций и во все выходные позиции перехода добавляется управляющая фишка. В выходных позициях перехода должна уже находиться или появится в этом же такте работы сети семантическая фишка. При наличии в позиции семантической и управляющей фишки они соединяются в единую фишку. Дополнительные условия срабатывания могут накладываться на переход по управлению при использовании при распараллеливании конструкций типа nowait, то есть, когда потоки завершаются и достаточно одного завершенного потока для продолжения работы, остальные в это время могут продолжать свое исполнение. В этом случае, наличия хотя бы одной объединенной фишки во входных позициях перехода join достаточно для его срабатывания. После первого срабатывания перехода, он работает только на извлечение фишек из входных позиций, без их добавления во входную.

При разделении объединенной фишки создается всегда одна управляющая фишка и столько семантических фишек, в скольких переходах по семантическим связям во входной функции находится данная позиция. Для конечных вычислительных операций, данные от которых больше нигде не используются, может быть не создано ни одной семантической фишки. Переходы по семантическим связям требуют, чтобы во всех входных позициях перехода была семантическая фишка, если иное не установлено в векторе условий срабатывания перехода Λ^S. Однако, существует дополнительное условие к семантической фишке. Она должна непременно быть результатом расщепления объединенной фишки, проверка чего осуществляется путем активации соответствующей дуги к переходу по семантическим связям только после срабатывания перехода по управлению, для которого эта же позиция является входной. Можно говорить о блокировке фишки, когда переход z_k^S не находится в возбуждённом состоянии, но на некоторых из его входных позиций $\exists a_j \in I_A(z_k^S)$ имеются фишки. Такой переход z_k^S называется блокирующим. Блокированная фишка не может участвовать во взаимодействии ни с какими другими переходами по семантическим связям кроме блокирующего. В единственную выходную позицию перехода после срабатывания помещается семантическая фишка, требующая объединения. Наличие дополнительных условий срабатывания перехода вызвано тем, что любая позиция не может принадлежать выходной функции более чем одного перехода по семантическим связям. Тогда при наличии циклических конструкций или ветвлений необходимо дополнительное условие «исключающего или» для ряда входных позиций.

Формирование и разделение объединенной фишки является больше абстрактным процессом. Каждый тип переходов изымает из вершины фишку исключительно своего типа и не пропускает «чужие» фишки.

Такая модель может эффективно использоваться как для моделирования, так и для верификации параллельных программ. В части верификации можно обнаружить следующие ошибки:

- некорректная последовательность действий, когда в позицию сперва попадает управляющая фишка;

- наличие гонок данных, то есть ситуаций, когда два или более параллельно работающих потока обращаются для записи к одной и той же переменной без использования синхронизации;
- наличие взаимных блокировок потоков, то есть ситуаций, когда потоки, получив эксклюзивные права на продолжение работы путем захвата блокировки, не могут её продолжить в связи с невозможностью получить следующую блокировку, захваченную другим потоком.

Анализ гонок данных в рамках РСПСС подразумевает проверку наличия семантических связей между параллельными участками сети при отсутствии синхронизации доступа между ними, который можно определить путем анализа связей по управлению. Определить это можно по наличию нескольких семантических зависимостей, где общая переменная является выходом (запись) или входом (чтение), а места её использования (входы и выходы переходов соответственно) относятся к параллельным участкам. При этом следует исключать из анализа переход z_i^S, где позиция a_i является входом, если его выход a_j является входом для перехода z_j^S, выходом которого является a_i, так как в данном случае переход z_i^S показывает не чтение, а использование памяти в операции записи. Наглядно это можно рассмотреть на примере позиции 0 рисунке 2, где семантические переходы 0 и 2 показывают использование памяти, с которой ассоциируется данная позиция, в операциях записи в неё, а семантические переходы 1 и 3 — зависимость памяти от операций записи в неё, и, так как позиции 3 и 5, являющиеся входами семантических переходов 1 и 3 соответственно, относятся к параллельным участкам, можно утверждать о наличии в модели гонки данных типа «запись — запись».

Для анализа взаимных блокировок, необходимо оценить достижимость соответствующей разметки сети, отображающей успешное выполнение этого параллельного участка, в случае наличия хотя бы одного пути на графе достижимости, который не приводит к желаемой разметке, определяется возможность возникновения взаимной блокировки (тупика). Более подробное рассмотрение этих задач не является целью данной статьи. Здесь хотелось бы обратить внимание, что в обоих случаях необходимым является анализ используемых механизмов синхронизации для обеспечения корректного доступа к переменным и вычислительным ресурсам системы. В связи с множеством различных используемых в разных технологиях распараллеливания механизмов синхронизации, необходимо определить их основные особенности и такие способы их корректного представления в РСПСС, которые позволят в дальнейшем автоматически строить их на основе кода параллельной программы и анализировать созданную сеть на предмет ошибок. В других работах, представляющих методы верификации параллельных программ на основе других вариантов сетей Петри, рассматривается лишь ограниченный набор примитивов синхронизации. Например, в работе [27] представлены только атомарные операции и мьютексы, в работе [30] — только условные переменные. Нашей целью является обобщение информации по всем основным способом синхронизации и представление их в виде нашей версии сетей Петри.

3. Основные типы операций синхронизации в параллельных программах

Для решения различных задач обеспечения взаимоисключающего доступа к общим данных в параллельных программах используют следующие основные методы синхронизации:

Атомарные операции — это такие операции, которые выполняются за один раз либо вообще не выполняются. Таким образом, если атомарная операция была запущена, она гарантирует, что дойдет до конца исполнения. В ряде языков программирования существуют средства задания и/или использования атомарных операций (например, класс Interlocked в .Net или директива atomic в OpenMP).

Барьер — это такой метод синхронизации, когда каждый поток из группы параллельных потоков, доходя до определённой точки программы (барьера), должен приостановить свою работу до тех пор, пока вся группа не достигнет заданной точки. Зачастую барьеры устанавливаются, когда необ-

ходимо получить промежуточные результаты вычислений, на основе которых будут выполняться следующие за барьером вычисления.

Критические секции представляют объекты синхронизации потоков, позволяющий предотвратить одновременное выполнение некоторого набора операций несколькими потоками. В один момент времени только один поток может исполнять код, принадлежащий соответствующей критической секции, все другие потоки, желающие попасть в критическую секцию, блокируются. Во многих технологиях параллельного программирования критические секции оформляются с использованием команд входа и выхода из критической секции (например, EnterCriticalSection, LeaveCriticalSection в WinAPI). Критические секции могут быть именованные, и в этом случае вхождение в одну критическую секцию, не оказывает влияние на доступ к другим.

Мьютекс — примитив синхронизации, обеспечивающий взаимное исключение критических участков кода. Несмотря на большое количество типов мьютексов в различных технологиях параллельного программирования (например, recursive_mutex, timed_mutex в std::thread, spin_mutex, queuing_mutex и т. д. в oneAPI TBB), каждый из которых обладает своими особенностями, принципиальный подход к их моделированию будет схож, но в некоторых случаях будет меняться порядок доступа к критическому коду потоков. Поэтому в данной статье будем рассматривать только классический мьютекс. Классический мьютекс представляется в виде переменной, которая может находиться в двух состояниях: в заблокированном и в незаблокированном. Когда нам нужно обеспечить взаимное исключение для некоторого кода потока, он вызывает блокирующую функцию мьютекса. Другие потоки, пытающиеся заблокировать тот же мьютекс, блокируются до тех пор, пока мьютекс не будет освобожден потоком, которому он принадлежит. После выполнения критического кода поток вызывает функцию для перевода мьютекса в разблокированное состояние и в то же время заблокированный этим мьютексом поток может продолжить свои вычисления. Если таких потоков много, планировщик выбирает один из них для возобновления выполнения.

Семафор является примитивом синхронизации, который позволяет ограничить количество потоков, которые имеют доступ к определённым ресурсам. Семафор представляет собой целую переменную, принимающую неотрицательные значения. Доступ к этой переменной осуществляется через две атомарные операции $P(S)$: пока $S == 0$ процесс блокируется; $S = S - 1$ и $V(S) : S = S + 1$. Во время операции P, если значение семафора больше нуля, то оно уменьшается на единицу, после чего можно получить доступ к критическому коду. Когда $S = 0$, поток блокируется до тех пор, пока какой-либо другой поток не увеличит значение S. Семафор может быть инициализирован начальным значением. Оно может быть равно нулю, что изначально запрещает доступ к любым данным, пока один из потоков не выполнит и не вызовет операцию V, или Любому другому неотрицательному значению N, что разрешает доступ к соответствующим данным несколько раз даже без выполнения операции V.

Условные переменные представляют примитив синхронизации, который обеспечивает блокировку одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. При этом, часто данные переменные дополнительно защищаются блокировками. При работе с условными переменными используются методы wait — для блокировки потока в ожидании условия, и метод signal (notify в std::thread) для информирования ожидающих потоков о срабатывании условия. В зависимости от реализации после операции signal может быть разблокирован один или несколько потоков.

Мониторы представляют собой высокоуровневый механизм взаимодействия и синхронизации процессов. Они представляют специальный тип данных, аналогичный классу или объекту в ООП, служащий для разграничения доступа к критическому коду программы. Монитор обладает собственными переменными, определяющими его состояния, а также функциями-методами, которые

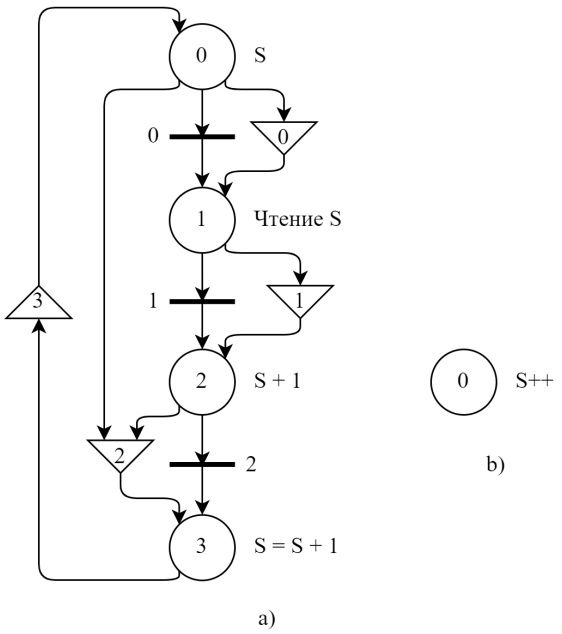

Fig. 3. Increment operation: a — non-atomic; b — atomic

Рис. 3. Операция инкремента: a — неатомарная; b — атомарная

могут изменять значения этих переменных и могут использовать только данные, находящиеся внутри монитора и свои параметры. Особенностью мониторов является то, что только один процесс может быть активен внутри данного монитора в любой момент времени. Фактически, с точки зрения задач моделирования мониторы представляют собой комбинацию уже рассмотренных выше примитивов синхронизации, а именно: мьютексов и условных переменных. Поэтому, в данной статье не будем останавливаться на них подробно.

4. Представление операций синхронизации в расширенных сетях Петри с семантическими связями

В этом разделе описывается представление в РСПСС вышеупомянутых примитивов синхронизации.

Атомарные операции. Простейшим примитивом является атомарная операция. Атомарная операция представляется как объединение нескольких операций в одной позиции РСПСС. Например, рассмотрим операцию увеличения семафорного счётчика $V(S) : S = S + 1$. Если бы она не был атомарной, то была бы представлена тремя позициями РСПСС (часть *a* на рисунке 3), но из-за атомарности это одна позиция (часть *b* на рисунке 3).

Для представления атомарной операции в первую очередь необходимо иметь РСПСС для программного кода без синхронизации, а затем реорганизовать его. Все переходы по семантическим связям, для которых изначальные позиции $A_{comb} = \left\{a_{comb_i}\right\}$, не объединённые еще в атомарную операцию, были выходными, объединяются в один переход. Множество входных позиций $I_A^S(z_{new}^S)$ нового перехода по семантическим связям z_{new}^S получается объединением множеств входных позиций объединяемых переходов $I_A^S(z_{old_i}^S)$ за исключением позиций, вошедших в новую атомарную, а выходная функция нового перехода $O_A^S(z_{new}^S)$ является позицией, представляющей атомарную операцию:

$$I_A^S(z_{new}^S) = \bigcup_i I_A^S(z_{old_i}^S) \backslash A_{comb}, O_A^S(z_{new}^S) = a_{atomic}.$$

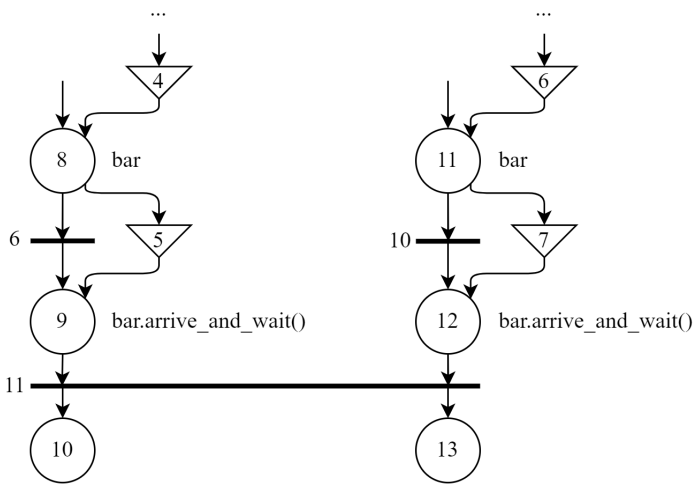

Fig. 4. Example of the barrier

Рис. 4. Пример барьерной синхронизации

Если какая-либо позиция a_i из множества A_{comb} являлась входной позицией любого иного перехода по семантическим связям, т. е. $I_A^S(z_j^S)$, то она должна быть заменена на новую атомарную позицию a_{atomic}.

$$I'^S_{A\_j} = \begin{cases} I_A^S(z_j^S), \text{ если } A_{comb} \cap I_A^S = \varnothing, \\ I_A^S(z_j^S) \cup \{a_{atomic}\} \setminus A_{comb}, \text{ иначе.} \end{cases}$$

Переходы по управлению между объединяемыми позициями должны быть удалены из сети. Мы сохраняем только переходы Z^C_{input}, которые вели в первую объединяемую позицию, и Z^C_{output}, которые имели последнюю объединяемую позицию в своей выходной функции. Больше ничего в РСПСС изменять не следует.

Барьеры в РСПСС представлены непримитивным переходом по управлению типа synchro. Этот тип переходов характеризуется тем, что он может иметь несколько входных и несколько выходных позиций. Для моделирования барьера необходимо определить точку в каждом потоке, после чего будет активирован режим ожидания всех потоков, которые соединены этим барьером. Эта точка представляет собой операцию вызова барьера на соответствующем языке программирования. Все такие позиции $a_{barrier_i}$ представляют входную функцию перехода synchro $I^C_{A\_synchro}$. Выходная функция перехода synchro $O^C_{A\_synchro}$ будет представлена позициями в каждом синхронизированном потоке, следующими за функцией барьера:

$$I^C_{A\_synchro} = \bigcup_i \left\{a_{barrier_i}\right\}, O^C_{A\_synchro} == \bigcup_i \left\{a_{barrier_i\_next}\right\}.$$

Семантические отношения не меняются при барьерной синхронизации. Пример барьерной синхронизации представлен на рисунке 4.

Как видно, имеются два потока, имеющие доступ к одному и тому же барьеру bar (позиции 8 и 11 на рисунке 4 — чтение памяти, хранящей барьер). Они вызывают функцию arrive_and_wait (позиции 9 и 12 на рисунке 4), чтобы остановить поток, пока другой не достигнет этой точки, после чего потоки продолжат свои вычисления (позиции 10 и 13 на рисунке 4).

Мьютекс — это особая позиция в РСПСС, отражающая состояние счётчика мьютекса (позиция lock на рисунке 5). Мьютекс не содержит фишку в позиции в заблокированном состоянии и содержит одну в разблокированном состоянии. Позиция счётчика является входом для переходов

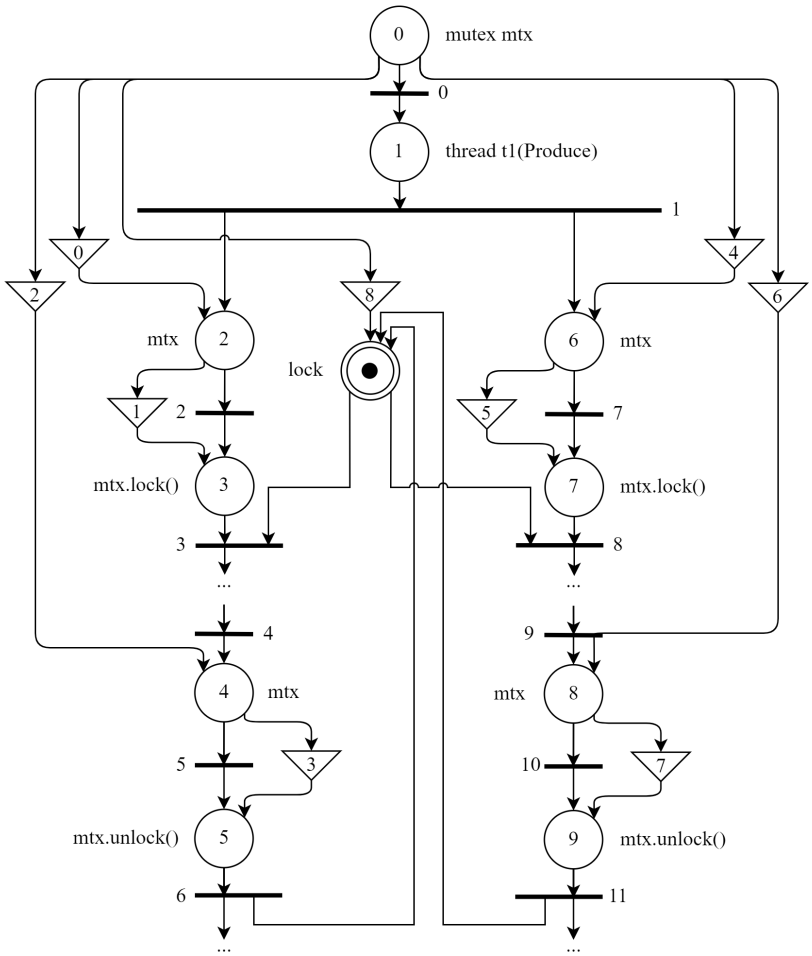

Fig. 5. Mutex representation in ExPNSR

Рис. 5. Представление мьютекса в РСПСС

по управлению, для которых другой позицией входа является операция попытки блокировки мьютекса (позиции 3 и 7 на рисунке 5) в соответствующем потоке. Поскольку в позиции счётчика мьютекса может быть только одна фишка, только один поток может продолжать свое выполнение. Переход по управлению после позиции освобождения мьютекса (позиции 5 и 9 на рисунке 5) имеет позицию счётчика в качестве одной из своих выходных позиций, что позволяет перевести его в разблокированное состояние и позволить другому потоку получить доступ к критичному коду.

Семантические связи внутри значимых операций не меняются. Позиция счётчика семантически зависит от позиции инициализации мьютекса (позиция 0, переход по семантическим связям 8 на рисунке 5). От одной и той же позиции инициализации зависят операции чтения состояния мьютекса (позиции 2, 4, 6 и 8 на рисунке 5 и переходы по семантическим связям 0, 2, 4, 6 соответственно) перед изменением его состояния.

Критические секции представляются аналогично мьютексам. Для каждой критической секции создается своя позиция, указывающая, находится ли какой-либо процесс в её критической секции или нет. Эта позиция идентична позиции счётчика мьютекса. Как и в случае с мьютексами, фишка в позиции означает, что поток может попасть в свою критическую секцию, а её отсутствие блокирует все остальные потоки. В этом случае позицией, управляющей изменением состояния критической секции, будет операция EnterCriticalSection как аналог операции блокировки мьютекса и LeaveCriticalSection как аналог операции разблокировки (освобождения) мьютекса. Операция

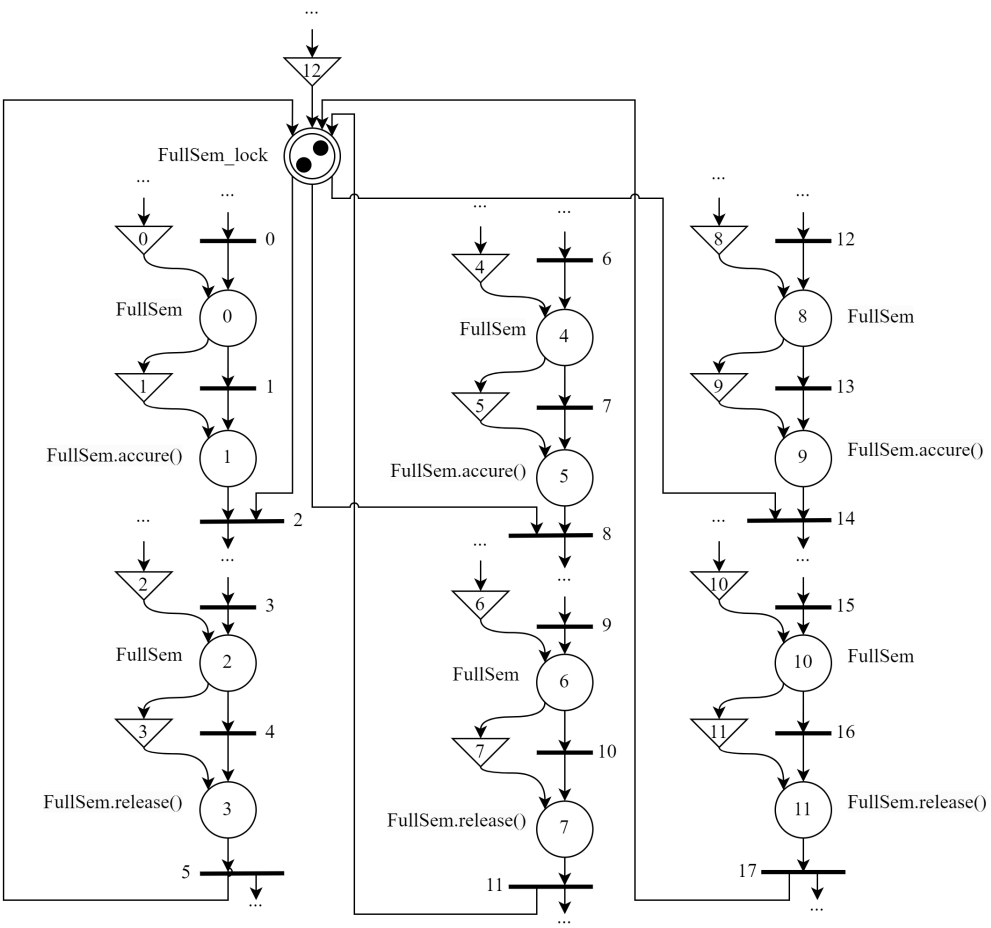

Fig. 6. Semaphore with several tokens in counter place in ExPNSR

Рис. 6. Семафор с несколькими фишками в позиции счётчика в РСПСС

InitializeCriticalSection играет ту же роль, что и инициализация мьютекса, и представлена в РСПСС таким же образом. Из-за того, что представление критической секции и мьютекса отличается только названием операций, мы не будем приводить в этой статье какой-либо дополнительный рисунок в качестве примера критической секции.

Моделирование **семафоров** также аналогично представлению мьютексов. Отличие заключается только в количестве фишек, которые могут быть в позиции семафора. Семафор, таким образом, является единственным вариантом позиции в РСПСС, когда нарушается правило наличия не более чем одной фишки в позиции. Такое возможно для теории РСПСС, так как в данном случае позиция не представляет операцию, а только регулирует запуск параллельных потоков. Таким образом, может быть сформулировано дополнительное правило для проверки модели параллельной программы: если в позиции оказалось более одной фишки, и эта позиция не является позицией семафора, то при моделировании программы была допущена ошибка. Само значение семафорной переменной S будет представлено в сети фишками в позиции счётчика (позиция FullSem_lock на рисунке 6). При этом при формировании исходной разметки РСПСС на месте счётчика сохраняется количество меток, равное начальному значению семафора.

Кроме того, значение счётчика семафора изначально может быть равно 0, что соответствует отсутствию фишки в позиции счётчика (позиция EmptySem_lock на рисунке 7). Это возможно,

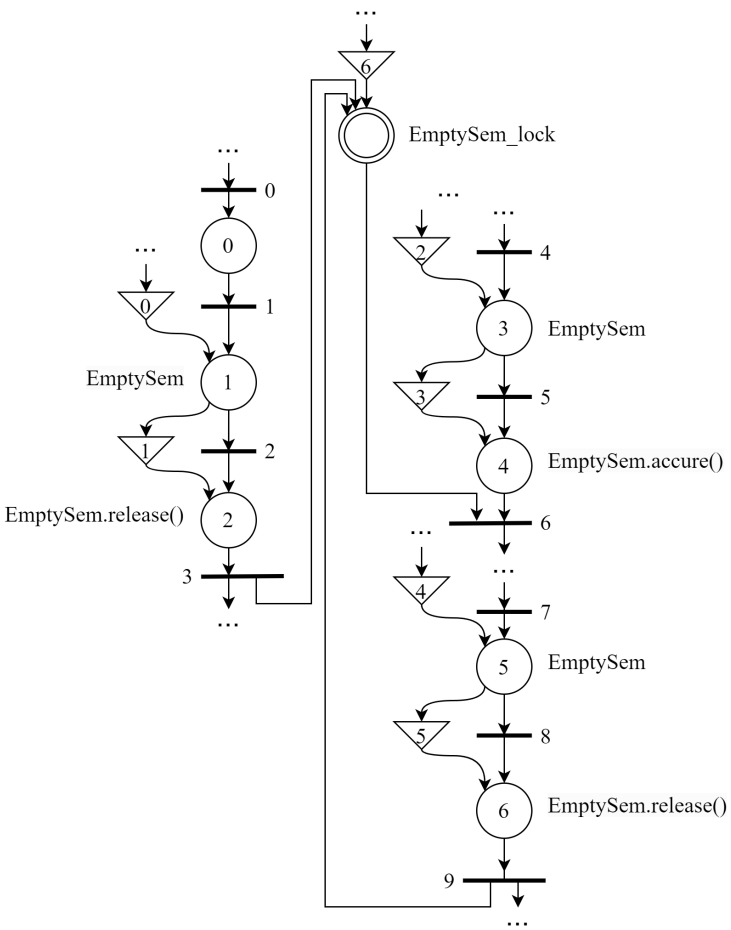

Fig. 7. Semaphore without tokens in counter place in ExPNSR

Рис. 7. Семафор без меток в позиции счётчика в РСПСС

потому что, в отличие от мьютекса, который должен быть разблокирован тем же потоком, который его заблокировал, операции с семафором могут выполняться независимо в разных потоках.

Для реализации некоторых сценариев работы программы может потребоваться, чтобы один из потоков дождался наступления определённых условий. Чтобы избежать постоянных проверок, поток можно приостановить до получения специального сигнала с помощью **условной переменной**. Один из потоков, использующий условную переменную, блокирует себя до тех пор, пока другой поток не освободит её (рисунок 8).

В зависимости от используемой технологии параллельного программирования могут быть небольшие особенности моделирования условных переменных. Мы рассмотрим реализацию, предоставляемую технологией std::thread. Обычно для целей блокировки используется unique_lock (позиции 0 и 6 на рисунке 8), который управляет блокировкой и разблокировкой мьютекса, связанного с условной переменной. Если какое-то условие (позиция 10 на рисунке 8) не выполняется, мьютекс разблокируется и вызывается метод ожидания условной переменной (позиция 13 на рисунке 8). Он блокирует текущий поток до тех пор, пока другой поток не вызовет метод notify_one (позиция 3 на рисунке 8), который отправит сигнал одному из ожидающих потоков (позиция 14 на рисунке 8). После этого текущий поток снова блокирует мьютекс и проверяет состояние.

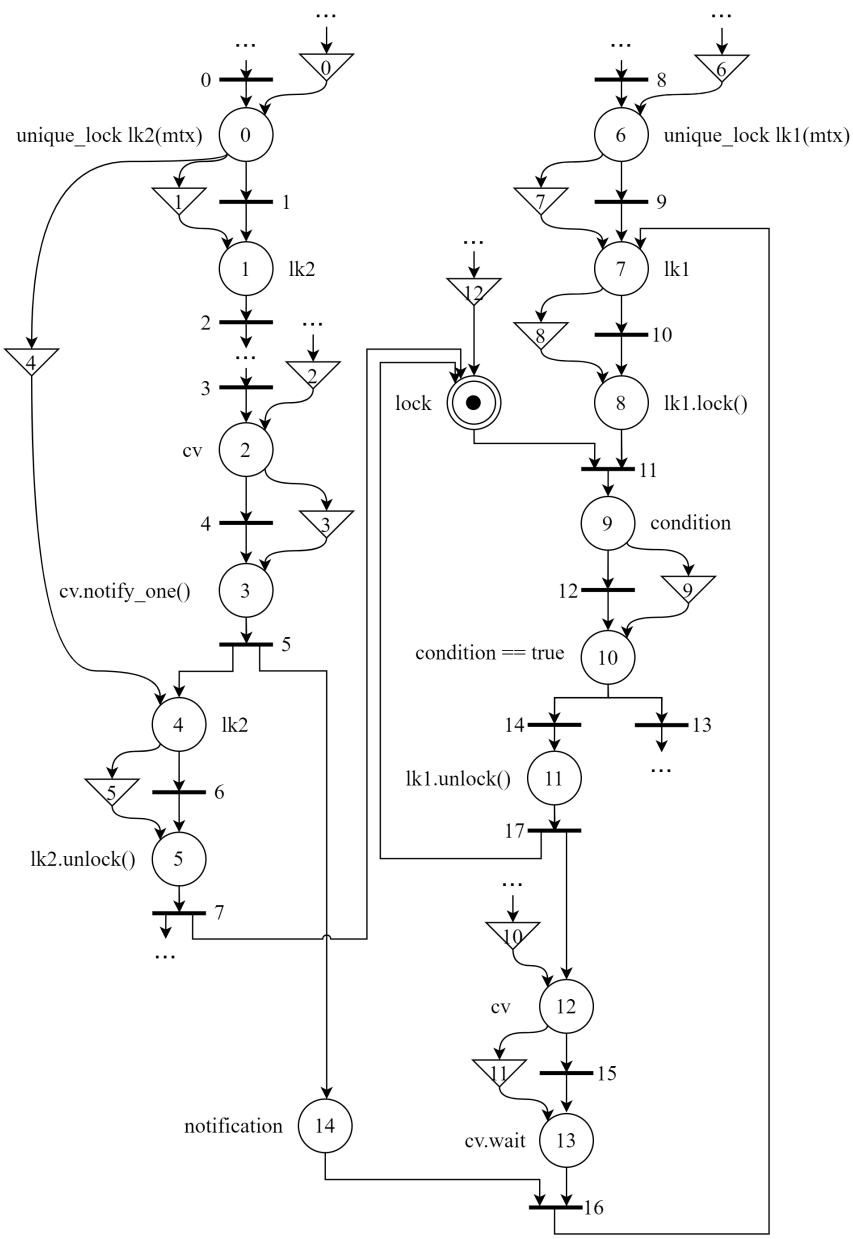

Fig. 8. Condition variable in ExPNSR

Рис. 8. Условные переменные в РСПСС

Как видно на рисунке 8, если условие в позиции 10 не выполнено, фишка возвращается в позицию счётчика мьютекса, а переход 16 не срабатывает до тех пор, пока в позиции 13 не появится фишка, что отражает наличие сигнала.

Представленные выше модели основных средств синхронизации могут использоваться в различных сочетаниях в реальных параллельных программах. И такое их представление позволяет определеть наличие и отсутствие ошибок синхронизации в параллельной программе. Вернемся к рисунку 2, где отсутствовала синхронизация между потоками и в результате между позициями 3 и 5 была обнаружена гонка данных типа «запись — запись». Наличие корректной синхронизации, например, с использованием мьютекса (рисунок 9) в таком случае не приведет к обнаружению такой ошибки.

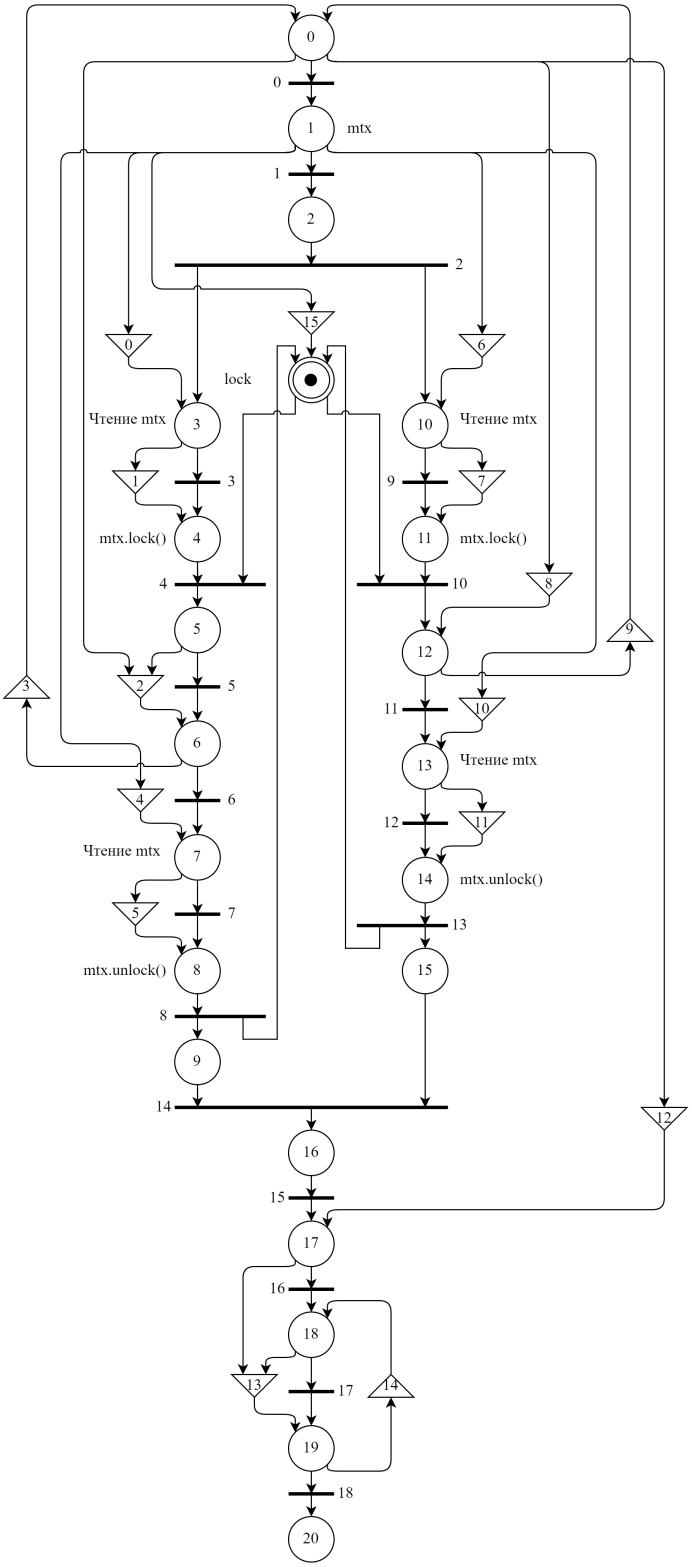

Fig. 9. ExPNSR of concurrent algorithm without data race

Рис. 9. РСПСС для параллельного алгоритма без гонки данных

Так как позиции 6 и 12, аналогичные позициям 3 и 5 рисунка 2 соответственно, находятся в критических секциях, образованных операциями блокировки мьютекса: позиции 4 и 11; соответствующие им операции будут выполняться последовательно, а значит гонка данных отсутствует.

Следует дополнительно отметить, что различные технологии распараллеливания могут иметь свои особенности реализации. Однако эти особенности не изменят принципов их моделирования, так как относятся к исполнению, а не к фундаментальной логике, которая проверяется на наличие ошибок. Эти характеристики лишь в некоторых случаях могут привести к добавлению каких-то позиций в РСПСС, или добавлению логических условий срабатывания переходов (например, для некоторых типов мьютексов).

Следует также отметить появление новых схем параллельной обработки, внедряемых в существующие языки (например, JEP 436, 437 для Java 20–21, Concurrency и Coroutines в C++ 20 и 23). Они требуют проведения статической верификации на более низком, системном уровне, что означает более сложный процесс обнаружения ошибок, стоимость которых при этом гораздо выше. В рамках реализации статического детектора тупиков и гонок данных для конкретных языков программирования и технологий распараллеливания это будет учтено в нашей дальнейшей работе.

Как уже говорилось ранее, такое представление программного кода в виде модели РСПСС позволяет автоматически формировать сеть, а также в дальнейшем её анализировать. Рассмотрим кратко как это осуществляется для языка Go. Построение модели идёт на основе уже имеющегося более простого, предлагаемого языком представления кода в виде абстрактного синтаксического дерева. Оно представляет собой конечное помеченное дерево, вершины которого соответствуют операторам языка, а листья — операндам (идентификаторам или константам), а основными элементами являются функции. Каждое объявление функции приводит к формированию новой РСПСС. Для каждой функции неявно создается позиция терминатора и переход по управлению из позиции, представляющей последнюю операцию данной функции, в позицию терминатора. Далее обрабатываются аргументы функции, для которых создаются соответствующие позиции сети. После чего происходит обработка операторов данной функции с учётом ключевых слов, позволяющих определить выделение памяти под переменные, обработку выражений, запуск нового потока, использование примитивов синхронизации и т. д. Для каждой операции формируются соответствующие позиции сети и в зависимости от их порядка следования в программе добавляются переходы по управлению. В случае запуска/синхронизации/завершения потока могут добавляться непримитивные переходы по управлению. После добавления новых позиций производится анализ, от каких данных зависит данная операция, находятся соответствующие позиции, порождающие эти данные, находящиеся до текущей позиции или в одном с ней параллельном блоке, и добавляются переходы по семантическим связям. Основным примитивом синхронизации в Go является мьютекс. В связи с наличием нескольких вариантов мьютексов, а также нескольких вариантов их объявления в программе разбор несколько усложняется, однако все равно позволяет найти такие объявления и сформировать в сети особую позицию мьютекса с предварительно оставленной фишкой в ней. Эта позиция сети связывается с позициями, определяющими операции блокировки и разблокировки мьютекса соответствующими переходами по управлению. Более подробно этот подход рассмотрен в работе [31].

Заключение

В статье были рассмотрены подходы к верификации параллельных программ, в частности поиск гонок данных и взаимных блокировок. Отмечено, что полное представление о наличие таких ошибок в программах могут дать только статические методы верификации. Поэтому авторами было решено использовать метод проверки модели, для которого предложено использовать собственную модификацию сетей Петри — расширенную сеть Петри с семантическими связями. Проблемы анализа параллельных программ на основе моделей заключаются в сложности их построения. В ранних работах было показано, как моделируются с использованием РСПСС последовательные алгоритмы,

в том числе с линейными, циклическими структурами и ветвлениями, а также параллельные программы с простейшим примитивом синхронизации — барьером. В данной статье подробно рассмотрены особенности моделирования различных используемых в наиболее часто применяемых технологиях параллельного программирования примитивов синхронизации. Предложенные в статье методы их моделирования помогут в дальнейшем в автоматизации представления параллельных программ в виде РСПСС и их анализа на наличие ошибок организации доступа к общим ресурсам. Дальнейшая работа направлена на обнаружение взаимоблокировок и гонок данных на основе анализа графа достижимости и семантических отношений в РСПСС.

References

[1] K. N. Zainidinov and Z. A. Karshiev, "Features of parallel execution of data mining algorithms", *Automatics and Software Enginery*, vol. 31, no. 1, pp. 83–91, 2020.

[2] C. Atilgan, B. T. Tezel, and E. Nasiboglu, "Efficient implementation and parallelization of fuzzy density based clustering", *Information Sciences*, vol. 575, pp. 454–467, 2021. DOI: 10.1016/j.ins.2021.06.044.

[3] A. Krizhevsky, *One weird trick for parallelizing convolutional neural networks*, 2014. arXiv: 1404.5997 [cs.NE].

[4] B. Blum and G. Gibson, "Stateless model checking with data-race preemption points", in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Association for Computing Machinery, 2016, pp. 477–493. DOI: 10.1145/2983990.2984036.

[5] A. N. Ivutin, A. G. Voloshko, and V. N. Izotov, "Approach to data race detection based on petri nets with additional semantic relations", in *Proceedings of the 2020 ELEKTRO*, 2020, pp. 1–5. DOI: 10.1109/ELEKTRO49696.2020.9130252.

[6] A. N. Ivutin, A. G. Troshina, and D. Yesikov, "Parallelization of algorithms with use the semantic Petri-Markov nets", *Vestnik of Ryazan State Radio Engineering University*, vol. 58, no. 4, pp. 49–56, 2016, in Russian. DOI: 10.21667/1995-4565-2016-58-4-49-56.

[7] A. V. Soloviev and D. A. Sedov, "Algorithms of detecting errors in parallel programs for distributed memory systems", in *Proceedings of the Instittute for Systems Analysis Russian Academy of Sciences (ISA RAS)*, in Russian, vol. 63, 2013, pp. 9–15.

[8] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, and H. Hansson, "Concurrency bugs in open source software: A case study", *Journal of Internet Services and Applications*, vol. 8, no. 1, pp. 4–19, 2017. DOI: 10.1186/s13174-017-0055-2.

[9] I. Tamas, I. Salomie, and M. Antal, "Atomic invariants verification and deadlock detection at compile-time", in *Proceedings of the IEEE 14th International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2018, pp. 435–441.

[10] Y. Cai, C. Ye, Q. Shi, and C. Zhang, "Peahen: Fast and precise static deadlock detection via context reduction", in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 784–796. DOI: 10.1145/3540250.3549110.

[11] M. Ronsse and K. De Bosschere, "RecPlay: A fully integrated practical record/replay system", *ACM Transactions on Computer Systems*, vol. 17, no. 2, pp. 133–152, 1999. DOI: 10.1145/312203.312214.

[12] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection", in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 121–133. DOI: 10.1145/1542476.1542490.

[13] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs", in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003, pp. 179–190. DOI: 10.1145/781498.781529.

[14] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice", in *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009, pp. 62–71. DOI: 10.1145/1791194.1791203.

[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs", *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997. DOI: 10.1145/265924.265927.

[16] X. Xie and J. Xue, "Acculock: Accurate and efficient detection of data races", in *Proceedings of the International Symposium on Code Generation and Optimization*, 2011, pp. 201–212. DOI: 10.1109/CGO.2011.5764688.

[17] M. Yu and D.-H. Bae, "SimpleLock+: Fast and accurate hybrid data race detection", *The Computer Journal*, vol. 59, no. 6, pp. 793–809, 2016. DOI: 10.1093/comjnl/bxu119.

[18] A. Legalov, V. Vasilyev, I. Matkovskii, and M. Ushakova, "Support tools for creation and transformation of functional-dataflow parallel programs", *Proceedings of the Institute for System Programming of the RAS*, vol. 29, no. 5, pp. 165–184, 2017. DOI: 10.15514/ISPRAS-2017-29(5)-10.

[19] V. Tikhomirov, S. Timofeev, and E. Moshkova, "Modified model checking verification method", *Bulletin of Russian Academy of Natural Sciences*, no. 3, pp. 118–121, 2018, in Russian.

[20] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, *et al.*, *Handbook of Model Checking*. Springer Cham, 2018, vol. 10.

[21] J. Barnat *et al.*, "Parallel model checking algorithms for linear-time temporal logic", in *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 457–507. DOI: 10.1007/978-3-319-63516-3_12.

[22] H. Boucheneb, G. Gardey, and O. H. Roux, "TCTL model checking of time Petri nets", *Journal of Logic and Computation*, vol. 19, no. 6, pp. 1509–1540, 2009. DOI: 10.1093/logcom/exp036.

[23] K. M. Kavi, A. Moshtaghi, and D.-j. Chen, "Modeling multithreaded applications using Petri nets", *International Journal of Parallel Programming*, vol. 30, pp. 353–371, 2002. DOI: 10.1023/A:1019917329895.

[24] G. Liu, M. Zhou, and C. Jiang, "Petri net models and collaborativeness for parallel processes with resource sharing and message passing", *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 4, pp. 1–20, 2017. DOI: 10.1145/2810001.

[25] S. Bandyopadhyay, D. Sarkar, and C. Mandal, "Equivalence checking of Petri net models of programs using static and dynamic cut-points", *Acta Informatica*, vol. 56, pp. 321–383, 2019. DOI: 10.1007/s00236-018-0320-2.

[26] A. Albaghajati and M. Ahmed, "CPN.Net: An automated colored Petri nets model extraction from .Net based source code", in *Proceedings of the 1st International Conference on Artificial Intelligence and Data Analytics (CAIDA)*, 2021, pp. 245–250. DOI: 10.1109/CAIDA51941.2021.9425201.

[27] M. Westergaard, "Verifying parallel algorithms and programs using coloured petri nets", in *Transactions on Petri Nets and Other Models of Concurrency VI*. Springer Berlin Heidelberg, 2012, pp. 146–168. DOI: 10.1007/978-3-642-35179-2_7.

[28] O. Kryukov, A. Voloshko, and A. Ivutin, "Automation of the formation of the maximum parallelism in the process", *Izvestiya Tul'skogo gosudarstvennogo universiteta. Tekhnicheskiye nauki*, vol. 10, pp. 230–237, 2022, in Russian.

[29] I. Lomazova, *Nested Petri nets: Modeling and analysis of distributedsystems with object structure*. Moscow: Scientific World, 2003. DOI: 10.13140/2.1.1843.9048.

[30] P. d. C. Gomes, D. Gurov, M. Huisman, and C. Artho, "Specification and verification of synchronization with condition variables", *Science of Computer Programming*, vol. 163, pp. 174–189, 2018. DOI: 10.1016/j.scico.2018.05.001.

[31] O. Kryukov, A. Voloshko, and A. Ivutin, "Method for automatically building a parallel program model in Go language", *Izvestiya Tul'skogo gosudarstvennogo universiteta. Tekhnicheskiye nauki*, vol. 2, pp. 420–433, 2025, in Russian.