УДК 004.4'242

МЕТОД ФОРМАЛЬНОЙ ВЕРИФИКАЦИИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ СЕТЕЙ ПЕТРИ

А. Н. Ивутин, к.т.н., доцент, заведующий кафедрой ВТ, ТулГУ, Тула, Россия;
orcid.org/ 0000-0003-2970-2148, e-mail: alexey.ivutin@gmail.com
А. Г. Трошина, к.т.н., доцент кафедры ВТ, ТулГУ, Тула, Россия;
orcid.org/ 0000-0002-4304-2513, e-mail: atroshina@mail.ru

Рассматривается задача формальной верификации параллельных программ с использованием математического аппарата теории сетей Петри. ***Целью работы*** *является формализация методов анализа параллельных программ для автоматизации их верификации. Корректный исходный последовательный программный код требует дополнительной верификации в связи с распределением команд и операций между параллельными потоками, их согласованием и синхронизацией. Результатом некорректного преобразования последовательного кода в параллельный является непредсказуемое поведение программы. Таким образом, основными задачами работы являются разработка методов верификации, а также исправления ошибок в параллельных программах. На основе теории сетей Петри с дополнительными семантическими связями сформированы и доказаны теоремы о соответствии параллельного алгоритма последовательному, разработаны методы исправления ошибок в параллельных алгоритмах.*

Ключевые слова: *сеть Петри, параллельная программа, эквивалентность сетей, верификация, семантические связи.*

DOI: 10.21667/1995-4565-2019-70-15-26

Введение

Параллельное программирование представляет значительную сложность, так как требует грамотной организации процесса взаимодействия параллельных потоков. При этом традиционные методы верификации программ [1] не могут быть применены. Дополнительная сложность при верификации параллельного кода заключается в множестве различных технологий распараллеливания, наиболее распространенными из которых являются POSIX Threads, Open MP, Intel TBB, MPI, а также многие другие, включая специальные языки параллельного программирования (например, HOPMA, ABCL, Adl, Erlang, Linda, Sisal) или функциональные языки, предоставляющие ряд возможностей для распараллеливания (например, Haskell и др.) [2]. В связи с этим можно выделить три основных направления развития методов верификации параллельных программ.

1. Привязанные к одной технологии (например, верификация для функциональных параллельных программ [3, 4], для технологии MPI [5, 6]), что делает их использование нерациональным в случае использования различных технологий параллельного программирования.

2. На основе промежуточного представления программы на особом языке, что стирает различия между различными технологиями распараллеливания [7], однако при автоматизации данной процедуры требуется разработка фронтендов, поддерживающих все или большинство различных технологий и языков параллельного программирования.

3. На основе разработки модели программы (например, конечно-автоматные модели (FSMs [8], EFSMs [9, 10], CFSMs [11, 12]), машины абстрактных состояний (ASMs [13, 14]), сети Петри [15, 16], сценарные подходы [Use cases [17], MSC и пр.], модели, основанные на различных логиках (пропозициональные логики и исчисления предикатов [18], временные логики [19], алгебры процессов и другие [20]) и ее проверки (например, методом проверки модели [21]). Хотя метод проверки модели широко используется и лежит в основе многих

инструментов, в том числе и BLAST [22], он обладает рядом недостатков, таких как «взрыв количества состояний» [23], а также требования к описанию спецификаций, которым должна удовлетворять параллельная программа, должны быть описаны на языке темпоральной логики, что требует дополнительной проверки корректности и полноты описания спецификаций.

Таким образом, наиболее перспективным подходом представляется использование комбинации промежуточного представления программы, которое не зависит от исходного языка программирования и моделирования параллельных программ. В качестве промежуточного представления предлагается использовать LLVM IR [24], а в качестве модели параллельной программы – математический аппарат расширенных сетей Петри – сетей Петри с дополнительными семантическими связями (СПДСС) [2], предоставляющий информацию не только о ходе протекания параллельного процесса, но и о семантических связях между операторами, что позволяет провести более глубокий анализ программы без ее запуска. Как показано в [25] такая сеть может быть автоматически получена из исходного программного кода. На основе двух автоматически построенных СПДСС – корректного последовательного кода и тестируемого параллельного кода может быть автоматически проведена верификация исследуемой программы. В данной работе формулируется метод анализа параллельных программ для автоматизации их верификации на основе математического аппарата СПДСС.

Математические основы разработки метода формальной верификации параллельных программ

Структура сети Петри с дополнительными семантическими связями (СПДСС) подробно описана в [26], здесь упомянем только основные элементы: A – конечное множество позиций; Z^C – конечное множество переходов по управлению; Z^S – конечное множество переходов по семантическим связям; $\tilde{R}^C$, $\tilde{R}^S$ – матрицы смежности размером, отображающие множество позиций в множество переходов по управлению и по семантическим связям соответственно; $\hat{R}^C$, $\hat{R}^S$ – матрица смежности размером, отображающая множество переходов по управлению и по семантическим связям соответственно в множество позиций. Позиции СПДСС являются математическими объектами, моделирующими процесс выполнения операции исполнительным устройством. Переход по управлению в СПДСС является математическим объектом, моделирующим процесс передачи управления по выполнению операций с одного процесса на другой. Переход по семантическим связям в СПДСС является математическим объектом, устанавливающим семантические ограничения на последовательность смены состояний одного или нескольких элементов системы.

Важным понятием аппарата СПДСС при моделировании информационного процесса является понятие фишки. Помещение в позицию фишки меняет состояние этих элементов структуры. Позиции, содержащие фишки (хотя бы одну), называют помеченными, а полное множество таких компонентов формирует разметку СПДСС. Позицию, содержащую фишку, (помеченную позицию) обозначим $a_{j(a,t)}$ (от англ. token).

Так как в сеть вводится еще один тип вершин – переходы по семантическим связям, то требуется изменение подхода к самой структуре фишек, а именно – формированию составной фишки, разделяемой на фишку (множество фишек) по управлению – T^C и фишку (множество фишек) по семантическим связям – T^S. Каждый из данных подтипов фишки формирует поведение соответствующего типа переходов, объединяясь в позициях в единую фишку $T = T^S + T^C$. Поведение семантической фишки отличается от фишки по управлению. Кардинальное число множества T^S всегда равно количеству переходов по семантическим связям, для которых данная вершина является входной функцией (либо 1 при отсутствии таких переходов), т.е. фактически означает, что в вершине производится клонирование фишки необходимое число раз: $|T^S| = 1 + \sum_{j=0}^{n} |a_{j(a)} = I_A(z^S_{j(z^S)})|$. Данное свойство не приводит к лавинообраз-

ному заполнению модели фишками в связи с запретом на принадлежность любой вершины более чем к одной выходной функции перехода по семантическим связям.

Кардинальное число множества T^C в каждой позиции (но не переходе) применительно к решаемой задаче всегда равно 1, что обусловлено физическими особенностями реализации алгоритмов исполнителями, поскольку один и тот же ресурс не может одновременно перерабатываться несколькими исполнителями.

Разметкой СПДСС М(П) называется функция, ставящая в соответствие каждой позиции $a_{j(a)} \in A$ целое неотрицательное число $M(a_{j(a)})$, определяющее количество фишек T в каждой позиции.

Утверждение 1. Операция получения единой фишки $T = T^S + T^C$ не является коммутативной.

Доказательство. Определение единой фишки как суммы двух фишек, где первая из них является семантической, означает, что первой в позиции (или одновременно) окажется именно семантическая фишка. Это вытекает из фундаментального свойства корректно спроектированного алгоритма, где для выполнения любой операции необходимо, чтобы вся априорная информация была получена. В случае, если первой приходит фишка по управлению, это означает, что необходимой информации нет, более того, в силу 1-ограниченности сети по управлению, взяться этой информации неоткуда, т.е. система является некорректной.

Формирование объединенной фишки является больше абстрактным процессом. Каждый тип переходов изымает из вершины фишку исключительно своего типа и не пропускает «чужие» фишки. Для систем рассматриваемого класса $M(a_{j(a)}) = \{0,1\}$. Процесс функционирования системы может быть представлен в виде последовательности смен состояний при реализации переходов по управлению. Последовательности состояний могут быть выстроены в одну из следующих структур: параллельную, линейную или с ветвлениями. Каждый компонент системы может быть разбит на такие элементы, последовательность состояний которых включает только примитивные переходы, а также линейные участки и участки с ветвлением.

Путь по управлению – это непустой сильно-связный граф $P^C = (A, Z^C, E)$, где $E = \{a_{1(a)} z^C_{1(z^C)}, z^C_{1(z^C)} a_{2(a)}, ..., z^C_{k(z^C)} a_{k(a)}\}$ и все $a_{j(a)}$ и $z^C_{j(z^C)}$ различны.

Путь по семантическим связям – это непустой сильно-связный граф $P^S = (A, Z^S, E)$, где $E = \{a_{1(a)} z^S_{1(z^S)}, z^S_{1(z^S)} a_{2(a)}, ..., z^S_{k(z^S)} a_{k(a)}\}$ и все $a_{j(a)}$ и $z^S_{j(z^S)}$ различны.

Если СПДСС представляет последовательный алгоритм по управлению, то множество путей по управлению из начальной позиции $a_{j(B)} \in A_B$ в конечную $a_{j(E)} \in A_E$ формируют множество альтернатив решения задачи $P^C_S = \{P^C_{S1}, P^C_{S2}, ..., P^C_{Sn}\}$.

Если СПДСС представляет параллельный алгоритм, то $P^C_{Si} = P^C_{a_{j(B)}, a_{j(E)}} \cup \bigcup_{p,q} P^C_{z^C_p, z^C_q}$, для которых выполняются условия: $P^C_{a_{j(B)}, a_{j(E)}} \cap P^C_{z^C_p, z^C_q} = z^C_p$, $|I_a(z^C_p)| = 1$, $|O_a(z^C_p)| > 1$.

Для каждой семантической связи существуют реализующие эту связь пути, где в начальном состоянии вырабатывается некоторый ресурс, необходимый последнему состоянию пути, причем ни одно состояние внутри пути не вырабатывает тот же ресурс. Одна семантическая связь может реализовываться по нескольким маршрутам.

Утверждение 2. Если между позициями $a_{i(a)} \in A$ и $a_{j(a)} \in A$ существует путь $P^S_{Sj} \in P^S_S$, то существует путь $P^C_{Sj} \in P^C_S$ между этими же позициями. Обратное утверждение не верно. Данное свойство вытекает из свойства связности графа по семантическим связям.

Утверждение 3. Если $a_{i(a)} \in A_E$, то $a_{i(a)} \notin I_A(z^S_{j(Z^S)})$ для всех j от 1 до J.

Наряду с определением путей по управлению и семантическим связям следует определить и отношения между позициями в сети, базирующиеся на основных свойствах, описанных ранее и характеризующих условия реализации алгоритма.

Между парой позиций $(a_{i(a)}, a_{ja})$ существует отношение предшествования по управлению $a_{i(a)} <_C a_{j(a)}$ или по семантическим связям $a_{i(a)} <_S a_{j(a)}$, если $a_{i(a)} \in I_A(z^C_{j(z^C)})$ и $a_{j(a)} \in O_A(z^C_{j(z^C)})$ или $a_{i(a)} \in I_A(z^S_{j(z^S)})$ и $a_{j(a)} \in O_A(z^S_{j(z^S)})$ соответственно, причем $a_{i(a)}$ называется предшественником по управлению своего преемника $a_{j(a)}$.

Отношения предшествования $<_C$ и $<_S$ транзитивны, а это значит, что имеют место следующие свойства: $a_{i(a)} <_{C(S)} a_{j(a)}, a_{j(a)} <_{C(S)} a_{k(a)} \Rightarrow a_{i(a)} <_{C(S)} a_{k(a)} \mid a_{i(a)}, a_{j(a)}, a_{k(a)} \in A$.

Утверждение 4. Если между парой позиций $(a_{i(a)}, a_{j(a)})$ существует отношение предшествования $a_{i(a)} <_S a_{j(a)}$, то переработка ресурса в $a_{j(a)}$ может начаться тогда и только тогда, когда переработка ресурса в $a_{i(a)}$ завершена.

Следствие 1. Если между парой позиций $(a_{i(a)}, a_{j(a)})$ существует отношение предшествования по семантическим связям $a_{i(a)} <_S a_{j(a)}$, то между ними существует и отношение предшествования по управлению $a_{i(a)} <_C a_{j(a)}$.

Следствие 2. Если между парой позиций $(a_{i(a)}, a_{j(a)})$ не существует отношения предшествования по семантическим связям $a_{i(a)} \not<_S a_{j(a)}$, то можно переформировать СПДСС, в которой не существует и отношение предшествования по управлению $a_{i(a)} \not<_C a_{j(a)}$.

Учитывая свойство транзитивности операции предшествования можно установить, что для предшествования по управлению между позициями $a_{i(a)}$ и $a_{j(a)}$ будет расположено некоторое количество позиций из множества A. Эта особенность и является ключевой для перестроения СПДСС, основываясь на семантических связях между позициями. При этом любое перестроение СПДСС не должно нарушать семантических связей в сети, а также состав множеств начальных $A_B \in A$ и конечных $A_E \in A$ позиций по управлению.

Утверждение 5. Если структура СПДСС такова, что для некоторых позиций $(a_{i(a)}, a_{j(a)} \in A)$, не являющихся branch-позицией, верно отношение предшествования по управлению $a_{i(a)} <_C a_{j(a)}$ и не выполняется отношение предшествования по семантическим связям $a_{i(a)} \not<_S a_{j(a)}$, то существует возможность перестроения СПДСС на основе семантической структуры сети, такая, что для любой пары позиций, связанных отношением предшествования по управлению, будут выполняться отношения предшествования и семантическим связям.

Утверждение 6. В любом корректно спроектированном алгоритме $\forall a_{j(a)} \mid (a_{j(a)} \in A_E) \exists a_{i(a)} <_C a_{j(a)} \mid (a_{i(a)} \in A_B)$. Верно и обратное утверждение.

Утверждение 6 говорит о том, что любой алгоритм, начинаясь в одной из начальных позиций, будет завершен в одной из конечных позиций.

Пара позиций $(a_{i(a)}, a_{j(a)})$ называется несовместимой по семантическим связям $a_{i(a)} \not<\!\!>_S a_{j(a)}$ (по управлению $a_{i(a)} \not<\!\!>_C a_{j(a)}$), если не существует отношения предшествования по семантическим связям (по управлению) ни между парой позиций $(a_{i(a)}, a_{j(a)})$, ни между парой позиций $(a_{j(a)}, a_{i(a)})$, т.е. не существует пути $P^S_{Si} \in P^S_S$ ($P^C_{Si} \in P^C_S$) такого, что выполняется условие $a_{i(a)} \in P^S_{Si}$ и $a_{j(a)} \in P^S_{Si}$ ($a_{i(a)} \in P^C_{Si}$ и $a_{j(a)} \in P^C_{Si}$).

Утверждение 7. В сети только с примитивными переходами по управлению для несовместимых по управлению позиций $a_{i(a)} \in A$ и $a_{j(a)} \in A$ существует позиция $a_{k(a)} \in A$, для которой имеет место следующее: $a_{i(a)} \not\gtrless_C a_{j(a)}, a_{k(a)} <_C a_{i(a)}, a_{k(a)} <_C a_{j(a)}, a_{i(a)}, a_{j(a)}, a_{k(a)} \in A$.

Можно сформулировать следующие свойства несовместимых позиций:

– в сети только с примитивными переходами по управлению несовместимые позиции по управлению не могут выполняться в рамках одного решения задачи;

– в сети только с примитивными переходами по управлению несовместимые позиции по управлению формируют альтернативные решения и принадлежат различным путям из branch-позиций.

Утверждение 8. Если известно, что две позиции являются несовместимыми по управлению, то они являются несовместимыми и по семантическим связям, т.е. $a_{i(a)} \not\gtrless_C a_{j(a)}$, $a_{i(a)} \not\gtrless_S a_{j(a)}$.

Доказательство. Используем доказательство от противного. Предположим, что для несовместимых по управлению позиций $a_{i(a)} \not\gtrless_C a_{j(a)}$ выполняется отношение предшествования по семантическим связям, например $a_{i(a)} <_S a_{j(a)}$. В соответствии с утверждением 4, если между позициями имеется отношение предшествования по семантическим связям $a_{i(a)} <_S a_{j(a)}$, то имеется и отношение предшествования по управлению $a_{i(a)} <_C a_{j(a)}$, что противоречит исходному условию.

Несовместимыми путями между двумя позициями по семантическим связям $P_j^S \not\gtrless_S P_i^S$ будем называть такие пути P_j^S и P_i^S, для которых выполняется условие $\forall a_{i(a)}, a_{j(a)} \mid a_{j(a)} \in P_i^S, a_{j(a)} \in P_j^S a_{i(a)} \not\gtrless_S a_{j(a)}$. Несовместимые позиции являются частным случаем несовместимых путей, то есть $P_i^S = \{a_{i(a)}\}$, $P_j^S = \{a_{j(a)}\}$ $a_{i(a)} \not\gtrless_S a_{j(a)}$. Очевидно, что исполняться параллельно могут только несовместимые пути.

Метод формальной верификации параллельных программ

Для формальной верификации параллельной программы необходимо иметь корректный последовательный алгоритм, на основе которого была (будет) разработана параллельная программа. Для верификации последовательного алгоритма предварительно могли быть применены любые классические методы верификации программ. Верификация параллельной программы строится на основании доказательства эквивалентности СПДСС для последовательного и параллельного кода, а также на основе проверки правильности эквивалентных преобразований.

Две сети Петри с дополнительными семантическими связями являются эквивалентными, если у них совпадают множества позиций и переходов по семантическим связям, и для каждого перехода по семантическим связям совпадают входные и выходные функции (условие эквивалентности сетей с точки зрения семантики), а также сохраняются отношения предшествования по управлению между позициями – началом ветвления, позициями, относящимися к ветвям ветвления, и позицией – объединения путей (завершающей позицией ветвления), т.е., если $a_{i(a)}$ – начальная позиция ветвления, $a_{j(a)}$ – завершающая позиция ветвления, $i \neq j$, а $b_1,...,b_n$ – позиции, относящиеся к ветвям ветвления, то должно сохраняться отношение предшествования по управлению $\forall k a_{i(a)} <_C b_k <_C a_{j(a)}$ (условие эквивалентности сетей с точки зрения управления).

$$\forall i, j \Pi_i = \{A_i, \{Z_i^C, \tilde{R}_i^C, \hat{R}_i^C\}, \{Z_i^S, \tilde{R}_i^S, \hat{R}_i^S\}\} \wedge \Pi_j = \{A_j, \{Z_j^C, \tilde{R}_j^C, \hat{R}_j^C\}, \{Z_j^S, \tilde{R}_j^S, \hat{R}_j^S\}\} \wedge$$

$$\wedge A_i = A_j \wedge Z_i^S = Z_j^S \wedge \tilde{R}_i^S = \tilde{R}_j^S \wedge \hat{R}_i^S = \hat{R}_j^S \wedge (\forall i(a), j(a) a_{i(a)} \in A_i \wedge a_{j(a)} \in A_j \wedge (\exists z_{i(z^C)}^C \in Z_i^C \wedge$$

$$\wedge a_{i(a)} \in I_A(z^C_{i(z^C)})) \wedge (\exists z^C_{j(z^C)} \in Z^C_i \wedge a_{i(a)} \in I_A(z^C_{j(z^C)})) \wedge (((\exists z^C_{k(z^C)} \in Z^C_i \wedge a_{j(a)} \in O_A(z^C_{k(z^C)})) \wedge$$
$$\wedge (\exists z^C_{l(z^C)} \in Z^C_i \wedge a_{j(a)} \in O_A(z^C_{l(z^C)}))) \vee (\exists z^C_{k(z^C)} \in Z^C_i \wedge a_{i(a)} \in O_A(z^C_{k(z^C)}) \wedge (\lambda(z^C_{k(z^C)}) = ...a_{k(a)} \vee a_{l(a)}...))) \wedge$$
$$\wedge (\forall kb_k \in A_i \wedge b_k \in A_j \wedge a_{i(a)} <_C b_k <_C a_{j(a)})) \Rightarrow \Pi_j \equiv \Pi_i .$$

Утверждение 9. Если в сетях, в которых отсутствуют непримитивные переходы по управлению, существуют две позиции $a_{i(a)}$ и $a_{j(a)}$, одна из которых $a_{i(a)}$ является входной функцией перехода по управлению $a_{i(a)} \in I_A(z^C_{j(z^C)})$, а другая $a_{j(a)}$ – выходной функцией этого же перехода $a_{j(a)} \in O_A(z^C_{j(z^C)})$ и эти позиции являются несовместимыми по семантическим связям $a_{i(a)} \not\gtrless_S a_{j(a)}$, и нет такого пути по управлению P^C_k, который содержит только одну позицию, но не содержит другую, то существует эквивалентная СПДСС, в которой $a_{j(a)}$ является входной функцией перехода по управлению s, а $a_{i(a)}$ выходной функцией этого же перехода: $a_{i(a)} \in O_A(z^C_{j(z^C)})$

$$\forall \Pi_i \mid \forall i(z^C) Z^C_{i(z^C)} = Z^C \wedge \forall z^C_{i(z^C)} \in Z^C_{i(z^C)} \left| I_A(z^C_{i(z^C)}) \right| = 1 \wedge \left| O_A(z^C_{i(z^C)}) \right| = 1, \forall i(a), j(a)(a_{i(a)} \in I_A(z^C_{i(z^C)}) \wedge$$
$$\wedge a_{j(a)} \in O_A(z^C_{i(z^C)}) \wedge a_{i(a)} \not\gtrless_S a_{j(a)} \wedge \nexists P^C_k \in P^C (a_{i(a)} \in P^C_k \wedge a_{j(a)} \notin P^C_k) \vee (a_{i(a)} \notin P^C_k \wedge$$
$$\wedge a_{j(a)} \in P^C_k)) \in \Pi_i \Rightarrow \exists \Pi_j \equiv \Pi_i (a_{i(a)} \in O_A(z^C_{i(z^C)}) \wedge a_{j(a)} \in I_A(z^C_{i(z^C)})) \in \Pi_j .$$

Утверждение 10. Если в сетях, в которых отсутствуют непримитивные переходы по управлению, существуют две позиции $a_{i(a)}$ и $a_{j(a)}$, одна из которых $a_{i(a)}$ является входной функцией перехода по управлению $a_{i(a)} \in I_A(z^C_{j(z^C)})$, а другая $a_{j(a)}$ – выходной функцией этого же перехода $a_{j(a)} \in O_A(z^C_{j(z^C)})$ и эти позиции являются несовместимыми по семантическим связям $a_{i(a)} \not\gtrless_S a_{j(a)}$, и существует позиция $a_{k(a)}$, для которой справедливо $a_{k(a)} \not<_S a_{i(a)}$ и $a_{j(a)} \not<_S a_{k(a)}$, и для любого пути по управлению P^C_k справедливо, что он содержит либо все позиции из тройки позиций $(a_{i(a)}, a_{j(a)}, a_{k(a)})$, либо не содержит ни одну из них, то существует эквивалентная СПДСС, в которой между позициями $a_{i(a)}$ и $a_{j(a)}$ может существовать более одного перехода по управлению, т.е. $a_{i(a)} <_C a_{k(a)} <_C a_{j(a)}$.

$$\forall \Pi_i \mid \forall i(z^C) Z^C_{i(z^C)} = Z^C \wedge \forall z^C_{i(z^C)} \in Z^C_{i(z^C)} \left| I_A(z^C_{i(z^C)}) \right| = 1 \wedge \left| O_A(z^C_{i(z^C)}) \right| = 1, \forall i(a), j(a)(a_{i(a)} \in I_A(z^C_{i(z^C)}) \wedge$$
$$\wedge a_{j(a)} \in O_A(z^C_{i(z^C)}) \wedge a_{i(a)} \not\gtrless_S a_{j(a)} \wedge a_{k(a)} \not<_S a_{i(a)} \wedge a_{j(a)} \not<_S a_{k(a)} \wedge \forall P^C_k \in P^C$$
$$(P^C_k \cap \{a_{i(a)}, a_{j(a)}, a_{k(a)}\} = \emptyset) \vee (P^C_k \cap \{a_{i(a)}, a_{j(a)}, a_{k(a)}\} = \{a_{i(a)}, a_{j(a)}, a_{k(a)}\})) \in \Pi_i \Rightarrow$$
$$\Rightarrow \exists \Pi_j \equiv \Pi_i (a_{i(a)} <_C a_{k(a)} <_C a_{ja)}) \in \Pi_j .$$

Утверждение 11. Если в сетях, в которых отсутствуют непримитивные переходы по управлению, существуют две позиции $a_{i(a)}$ и $a_{j(a)}$, одна из которых $a_{i(a)}$ является входной функцией перехода по управлению $a_{i(a)} \in I_A(z^C_{j(z^C)})$, а другая $a_{j(a)}$ – выходной функцией этого же перехода $a_{j(a)} \in O_A(z^C_{j(z^C)})$ и эти позиции являются несовместимыми по семантическим связям $a_{i(a)} \not\gtrless_S a_{j(a)}$ и нет такого пути по управлению P^C_k, который содержит только одну позицию, но не содержит другую, то существует эквивалентная СПДСС с непримитивными переходами типа fork, join, synchro, в которой отношение предшествования по управлению между позициями $a_{i(a)}$ и $a_{j(a)}$ может быть нарушено, то есть в которой пара позиций $(a_{i(a)}, a_{j(a)})$ несовместима по управлению $a_{i(a)} \not\gtrless_C a_{j(a)}$.

$$\forall \Pi_i \mid \forall i(z^C) Z^C_{i(z^C)} = Z^C \wedge \forall z^C_{i(z^C)} \in Z^C_{i(z^C)} \left| I_A(z^C_{i(z^C)}) \right| = 1 \wedge \left| O_A(z^C_{i(z^C)}) \right| = 1, \forall i(a), j(a)(a_{i(a)} \in I_A(z^C_{i(z^C)}) \wedge$$

$$\wedge a_{j(a)} \in O_A(z^C_{i(z^C)}) \wedge a_{i(a)} \not> \not<_S a_{j(a)} \wedge \exists P^C_k \in P^C (a_{i(a)} \in P^C_k \wedge a_{j(a)} \notin P^C_k) \vee (a_{i(a)} \notin P^C_k \wedge a_{j(a)} \in P^C_k))$$

$$\in \Pi_i \Rightarrow \exists \Pi_j \equiv \Pi_i (a_{i(a)} \not> \not<_C a_{j(a)} \wedge Z^C_{i(z^C)} = Z^C \wedge \exists z^C_{i(z^C)} \in Z^C_{i(z^C)} \left| I_A(z^C_{i(z^C)}) \right| > 1 \vee \left| O_A(z^C_{i(z^C)}) \right| > 1) \in \Pi_j .$$

Утверждение 12. Если в сетях, в которых отсутствуют непримитивные переходы по управлению, для любой позиции, из которой выходит минимум одна семантическая связь, всегда можно построить такую эквивалентную СПДСС, в которой между этой позицией и минимум одной из подмножества позиций, соединенных с ней семантической связью, будет не более одного перехода по управлению. Выражая это свойство через отношение предшествования, можем записать следующее:

$$\forall \Pi_i \mid \forall i(z^C) Z^C_{i(z^C)} = Z^C \wedge \forall z^C_{i(z^C)} \in Z^C_{i(z^C)} \left| I_A(z^C_{i(z^C)}) \right| = 1 \wedge \left| O_A(z^C_{i(z^C)}) \right| = 1, (\forall a_{i(a)} a_{i(a)} \in I_A(z^S_{i(z^S)}) \wedge$$

$$\wedge a_{j(a)} \in O_A(z^S_{i(z^S)})) \in \Pi_i \Rightarrow \exists \Pi_j \equiv \Pi_i (\exists z^C_{j(z^C)} a_{i(a)} <_C a_{j(a)} \wedge \nexists a_{k(a)} a_{i(a)} <_C a_{k(a)} <_C a_{j(a)}) \in \Pi_j .$$

Следствие. Для любой позиции, из которой выходит такая семантическая связь, для которой данная позиция является единственной входной, либо остальные позиции являются предшествующими по семантическим связям, можно построить такую эквивалентную СПДСС, в которой между данной позицией и последующими по семантическим связям будет расположен один переход, то есть

$$\forall \Pi_i \mid \forall a_{i(a)}((a_{i(a)} \in I_A(z^S_{i(z^S)})) \vee (a_{i(a)} \in I_A(z^S_{i(z^S)}) \wedge \forall a_{k(a)} \in I_A(z^S_{i(z^S)}) \wedge a_{k(a)} <_S a_{i(a)}) \wedge$$

$$\wedge (\{a_{j(a)}\} = O_A(z^S_{i(z^S)})))) \in \Pi_i \Rightarrow \exists \Pi_j \equiv \Pi_i (\exists z^C_{i(z^C)} \in Z^C \wedge a_{k(a)} \in I_A(z^C_{i(z^C)}) \wedge \{a_{j(a)}\} = O_A(z^C_{i(z^C)})) \in \Pi_j .$$

Одной из критических проблем при распараллеливании является проблема гонок данных [27], то есть ситуации, в которой результат работы приложения зависит от последовательности выполнения команд в параллельных потоках. Несмотря на то, что в последнее время развиваются методы статического анализа кода [28, 29], являющиеся более перспективными в сравнении с динамическими методами [30], абсолютной точности работы методов еще не удалось добиться.

Анализ наличия ситуации гонок данных требует модернизации СПДСС, а именно – добавления в математическое представление двух множеств: множества позиций, выполняющих выделение памяти под переменную – множества $Al = \{a_{i(a)}\}$, и множества позиций, выполняющих запись $W = \{a_{i(a)}\}$. В сети все позиции, осуществляющие чтение и запись в переменную, имеют семантическую связь с соответствующей позицией выделения памяти.

Утверждение 13. Если в параллельных потоках имеются позиции, осуществляющие запись в память, и позиции чтения или записи необходимо проследить, чтобы порядок их выполнения соответствовал порядку исходного последовательного алгоритма, то для этого необходимо, чтобы либо эти позиции должны выполняться последовательно в одном потоке, либо необходимо добавление перехода по управлению типа synchro.

$$a_{i(a)} \in W \wedge (a_{i(a)} <>_S a_{j(a)}) \wedge (a_{k(a)} <_S a_{i(a)}) \wedge (a_{k(a)} <_S a_{j(a)}) \wedge a_{k(a)} \in Al \wedge$$

$$\wedge (a_{i(a)} <_C a_{j(a)}) \in Seq \Rightarrow (a_{i(a)} <_C a_{j(a)}) \in Par.$$

Рассмотрим демонстрационный пример распараллеливания с учетом гонок данных. Это только демонстрация подхода, очевидно, что накладные расходы на создание, завершение и синхронизацию потоков слишком велики для такого кода. Ниже приведен последовательный код на языке C++ и его промежуточное представление LLVM, а также фрагмент параллельного кода [24], который в связи с особенностью организации работы с памятью значительно разрастается.

Сеть Петри с дополнительными семантическими связями выглядит следующим образом (рисунок 1, *а*). Отдельно следует отметить, что команда store, осуществляющая запись, рас-

сматривается неразрывно от команды, идущей непосредственно перед ней, или совместно с командой выделения памяти alloca. Такие команды отображаются в СПДСС одной позицией.

```
int main()          ; Function Attrs: noinline norecurse    ; Function Attrs: noinline norecurse opt-
{                   nounwind optnone ssp uwtable define     none uwtable
                    i32 @main() #0 {                        define i32 @main() #2 {…
int a = 10;           %1 = alloca i32, align 4                %22 = call i32 @pthread_create(i64* %2,
int b;                %2 = alloca i32, align 4              %union.pthread_attr_t* null, i8* (i8*)*
                      store i32 20, i32* %1, align 4        @_Z11threadFunc1Pv, i8* %21) #9
  b = a +             %3 = load i32, i32* %1, align 4         %23 = load %struct.th2ArgsS*,
10;                   %4 = add nsw i32 10, %3               %struct.th2ArgsS** %5, align 8
                      store i32 %4, i32* %2, align 4          %24 = bitcast %struct.th2ArgsS* %23 to
  a = a + 1;          %5 = load i32, i32* %1, align 4       i8*
                      %6 = add nsw i32 1, %5                  %25 = call i32 @pthread_create(i64* %3,
                      store i32 %6, i32* %1, align 4        %union.pthread_attr_t* null, i8* (i8*)*
                                                            @_Z11threadFunc2Pv, i8* %24) #9
  return 0; }         ret i32 0}                              %26 = load i64, i64* %2, align 8
                                                              %27 = call i32 @pthread_join(i64 %26,
                                                            i8** null)
                                                              %28 = load i64, i64* %3, align 8
                                                              %29 = call i32 @pthread_join(i64 %28,
                                                            i8** null)
                                                              ret i32 0}
```

Данную СПДСС можно описать с использованием следующих множеств: $A=\{a_1,a_2,a_3,a_4,a_5,a_6,a_7\}$, $Z^C=\{z_1^C,z_2^C,z_3^C,z_4^C,z_5^C,z_6^C\}$, $Z^S=\{z_1^S,z_2^S,z_3^S,z_4^S\}$, $Al=\{a_1,a_2\}$, $W=\{a_1,a_5,a_6\}$; и матриц $\widetilde{R}^S=\{\{a_1,a_1,a_1,a_2,a_3,a_5\},\ \{z_1^S,z_3^S,z_4^S,z_2^S,z_2^S,z_4^S\}\}$, $\widehat{R}^S=\{\{z_1^S,z_2^S,z_3^S,z_4^S\},\ \{a_3,a_4,a_5,a_6\}\}$, $\widetilde{R}^C=\{\{a_1,a_2,a_3,a_4,a_5,a_6\},\ \{z_1^C,z_2^C,z_3^C,z_4^C,z_5^C,z_6^C\}\}$, $\widehat{R}^C=\{\{z_1^C,z_2^C,z_3^C,z_4^C,z_5^C,z_6^C\},\ \{a_2,a_3,a_4,a_5,a_6,a_7\}\}$.

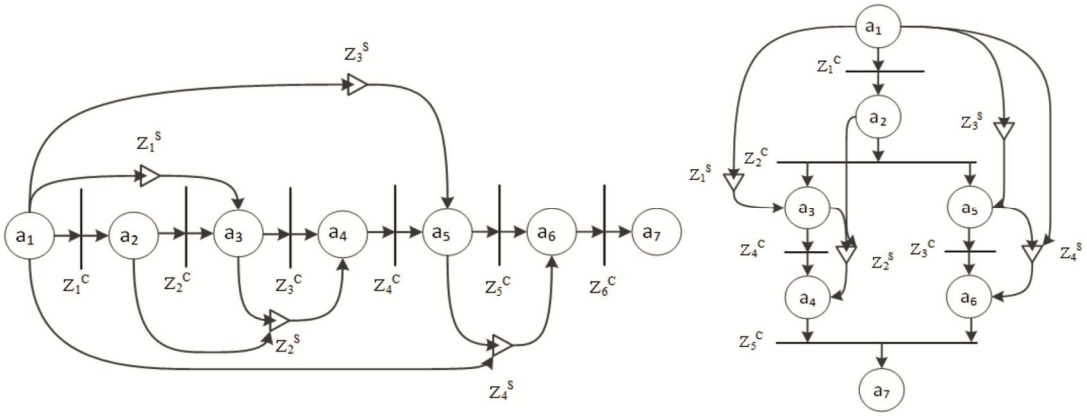
Распараллеленная без учета условий гонок данных СПДСС представлена на рисунке 1, *б*. После распараллеливания изменению подвергаются только множества переходов по управлению и функции инцидентности для переходов по управлению, то есть $Z^C=\{z_1^C,z_2^C,z_3^C,z_4^C,z_5^C\}$, $\widetilde{R}^C=\{\{a_1,a_2,a_3,a_4,a_5,a_6\},\ \{z_1^C,z_2^C,z_3^C,z_5^C,z_4^C,z_5^C\}\}$, $\widehat{R}^C=\{\{z_1^C,z_2^C,z_2^C,z_3^C,z_4^C,z_5^C,z_6^C\},\ \{a_2,a_3,a_5,a_4,a_6,a_7\}\}$. Таким образом, имеются позиции параллельных потоков a_3 и a_6, в первой из которых осуществляется чтение из переменной, а во второй запись в ту же переменную, а следовательно, в соответствии с утверждением 13, для устранения гонок данных при распараллеливании необходимо обеспечить выполнение этих позиций в том же порядке, как было в изначальном последовательном алгоритме. В данном случае распараллеливание программного кода не целесообразно.

На основании приведенных выше утверждений можно сформулировать следующий метод формальной верификации параллельных программ.

1. На первом этапе необходимо перевести изначальный программный код последовательного и параллельного алгоритмов в промежуточное представление. Далее анализу будут подвергаться только функции, отличающиеся между собой.

2. Исходя из промежуточного представления программных кодов могут быть автоматически сформированы СПДСС для последовательной и параллельной реализаций программы. В среднем одна функция на языке высокого уровня содержит 50 – 150 строк кода [31], а значит, промежуточное представление увеличит это значение примерно в 5 – 7 раз. Так как матрицы инцидентности по управлению и по семантическим связям являются сильно разреженными, то для их хранения предлагается использовать способ представления в виде двух век-

торов: в одном содержится номер строки, в котором присутствует единица, а в другом – номер столбца, в котором находится эта же единица. Такое представление позволяет алгоритму анализа эффективно использовать оперативную память и вычислительные ресурсы, так как требуется работа всего с четырьмя векторами средней длины.



а (a) **б (b)**

Рисунок 1 – СПДСС для последовательного (а) и распараллеленного (б) кода
Figure 1 – PNASR for sequential (a) and parallelized (b) code

3. Устанавливается соответствие между позициями последовательной и параллельной сети. Если для какой-либо позиции изначального алгоритма не удается установить такое соответствие, то формально данные две программы не являются эквивалентными. В этом случае можно выполнить дополнительные проверки такого алгоритма, однако в данной ситуации требуется дополнительная ручная проверка тех участков, которые связаны с новыми позициями в параллельном коде. Для этого формируется множество разности позиций параллельного и исходного кодов.

4. Проводится анализ семантических связей в СПДСС. В соответствии с определением эквивалентности сетей для каждого перехода по семантическим связям исходного последовательного алгоритма должен существовать переход по семантическим связям параллельного алгоритма, для которых полностью совпадают семантические связи. Если в СПДСС параллельного кода исчезла какая-либо позиция, которая была в исходном алгоритме, необходимо выполнить проверку семантических связей между оставшимися позициями и для всех позиций-последователей удаленной позиции должно сохраняться отношение предшествования с позициями – предшественниками удаленной позиции.

5. Анализ функционирования сети с точки зрения управления. Проверяется соответствие утверждения 2 для параллельного алгоритма, то есть наличие пути по управлению между позициями, связанными по семантике путем моделирования функционирования сети с использованием фишек.

6. Анализ корректности формирования переходов fork, join synchro в соответствии с утверждениями 4 – 7.

7. Анализ параллельных потоков на отсутствие семантических связей между позициями, принадлежащими разным потокам между переходами fork-synchro, synchro-synchro, synchro-join и fork-join. Данный этап также включает в себя анализ необходимости включения дополнительных переходов типа synchro, то есть анализ программы на блокировки.

8. Анализ параллельных потоков на наличие гонок данных.

Заключение

В статье предложена методика верификации параллельных алгоритмов на основании разработанной концепции расширенных сетей Петри – сетей Петри с дополнительными семантическими связями. Введены правила для проверки эквивалентности сетей и правила поиска и исключения состояния гонок данных.

Исследование выполнено при финансовой поддержке РФФИ и Правительства Тульской области в рамках научного проекта 19-41-710003 р_а.

Библиографический список

1. **Кулямин В. В.** Методы верификации программного обеспечения // Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению «Информационно-телекоммуникационные системы». 2008. Т. 117.

2. **Ивутин А. Н., Трошина А. Г., Есиков Д. О.** Применение семантических сетей Петри-Маркова для решения задачи распараллеливания алгоритмов // Вестник Рязанского государственного радиотехнического университета. Рязань: РГРТУ. 2016. № 58. С. 49-56.

3. **Легалов А. И.** Функциональный язык для создания архитектурно-независимых параллельных программ //Вычислительные технологии. 2005. Т. 10. №. 1.

4. **Кропачева М. С., Легалов А. И.** Формальная верификация программ, написанных на функционально-потоковом языке параллельного программирования // Моделирование и анализ информационных систем. 2015. Т. 19. №. 5. С. 81-99.

5. **Siegel S. F., Zirkel T. K.** Automatic formal verification of MPI-based parallel programs // ACM Sigplan Notices. 2011, vol. 46, no. 8, pp. 309-310.

6. **Афанасьев К. Е., Власенко А. Ю.** Семантические ошибки в параллельных программах для систем с распределенной памятью и методы их обнаружения современными средствами отладки // Вестник Кемеровского государственного университета. 2009. №. 2.

7. **Кулямин В. В.** Интеграция методов верификации программных систем // Программирование. 2009. Т. 35. №. 4. С. 41-55.

8. **Хопкрофт Д. Э., Мотвани Р., Ульман Д.** Введение в теорию автоматов, языков и вычислений. 2008.

9. **G. A. Simon.** An extended finite state machine approach to protocol specification. Proc. of 2-nd International Workshop on Protocol Specification, Testing and Verification, pp. 113-133, North-Holland Publishing Co., 1982.

10. **Cheng K. T., Krishnakumar A. S.** Automatic functional test generation using the extended finite state machine model //30th ACM/IEEE Design Automation Conference. IEEE, 1993, pp. 86-91.

11. **Luo G., von Bochmann G., Petrenko A.** Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method //IEEE Transactions on Software Engineering. 1994. Т. 20. №. 2. С. 149-162.

12. **Balarin F. et al.** Formal verification of embedded systems based on CFSM networks //33rd Design Automation Conference Proceedings, 1996. IEEE, 1996. С. 568-571.

13. **Gurevich Y.** Evolving Algebras: An Introductory Tutorial //Bull. EATCS. 1991, vol. 43, pp. 264-284.

14. **Гуревич Ю.** Последовательные машины абстрактных состояний охватывают последовательные алгоритмы //Системная информатика. 2004. Т. 9. С. 7-50.

15. **Питерсон Дж.** Теория сетей Петри и моделирование систем. М.: Мир, 1984. 264 с.

16. **Ramchandani C.** Analysis of asynchronous concurrent systems by timed Petri nets // PhD Thesis. Cambridge, Mass.: MIT, Dept. Electrical Engineering, 1974.

17. **Kösters G., Six H. W., Winter M.** Coupling use cases and class models as a means for validation and verification of requirements specifications // Requirements engineering. 2001, vol. 6, no. 1, pp. 3-17.

18. **Hoare C. A. R.** An axiomatic basis for computer programming // Communications of the ACM. – 1969, vol. 12, no. 10, pp. 576-580.

19. **Harel D., Kozen D., Tiuryn J.** Dynamic logic // Handbook of philosophical logic. Springer, Dordrecht, 2001. С. 99-217.

20. **Бурдонов И. Б. и др.** Обзор подходов к верификации распределенных систем //ИСП РАН, препринт. 2003. Т. 16.

21. **Grumberg O., Long D. E.** Model checking and modular verification // ACM Transactions on Programming Languages and Systems (TOPLAS). 1994, vol. 16, no. 3, pp. 843-871.

22. **Henzinger T. A., Jhala R., Majumdar R.** The BLAST software verification system // International SPIN Workshop on Model Checking of Software. Springer, Berlin, Heidelberg, 2005, pp. 25-26.

23. **Карпов Ю. Г.** MODEL CHECKING. Верификация параллельных и распределенных программных систем. БХВ-Петербург, 2010.

24. **Lattner C.** LLVM Language Reference Manual: LLVM compiler infrastructure // LLVM Project. 2003-2017. URL: http://llvm.org/docs/index.html (дата обращения: 30.08.2019).

25. **Ivutin A., Troshina A., Novikov A. S.** Low-level Code Auto-tuning for State-of-the-art Multicore Architectures // 2019 29th International Conference Radioelektronika (RADIOELEKTRONIKA). IEEE, 2019, pp. 1-6.

26. **Ivutin A. N., Troshina A. G.** Semantic Petri-Markov nets for automotive algorithms transformations // 2018 28th International Conference Radioelektronika (RADIOELEKTRONIKA). IEEE, 2018, pp. 1-6.

27. **Уильямс Э.** Параллельное программирование на C++ в действии. Практика разработки многопоточных программ / пер. с англ. А.В. Слинкин М.: ДМК Пресс, 2012. 672 с.

28. **Андрианов П. С., Мутилин В. С., Хорошилов А. В.** Метод легковесного статического анализа для поиска состояний гонок // Труды Института системного программирования РАН. 2015. Т. 27. №. 5.

29. **Прокопенко А. С.** Статический анализ условий гонки в параллельных программах на разделяемой памяти: дис. Москва, 2010. 107 с.

30. **Трифанов В. Ю., Цителов Д. И.** Динамические средства обнаружения гонок в параллельных программах // Компьютерные инструменты в образовании. 2011. №. 6.

31. **Макконнелл С.** Совершенный код. Мастер-класс. Пер. с англ. М.: Издательство «Русская редакция», 2010. 896 с.

UDC 004.4'242

PETRI-NET BASED METHOD OF PARALLEL PROGRAMS FORMAL VERIFICATION

A. N. Ivutin, Ph.D (Tech.), accociate professor, Head of the Department CT, TSU, Tula, Russia;
orcid.org/ 0000-0003-2970-2148, e-mail: alexey.ivutin@gmail.com
A. G. Troshina, Ph.D (Tech.), associate professor, CT department, TSU, Tula, Russia;
orcid.org/ 0000-0002-4304-2513, e-mail: atroshina@mail.ru

The problem of parallel programs formal verification using mathematical apparatus of the theory of Petri nets is considered. ***The aim of the work*** *is to formalize methods of parallel program analysis to automate their verification. Correct source sequential program code requires additional verification in connection with the distribution of commands and operations between parallel threads, their coordination and synchronization. The result of incorrect conversion of sequential code to parallel is unpredictable behavior of the program. Thus, the main objective of the work is the development of verification methods, as well as error correction in parallel programs. Based on the theory of Petri nets with additional semantic relations, theorems on the equivalence of parallel and sequential algorithms are formed and proved, error correction methods in parallel algorithms are developed.*

Key words: *Petri net, parallel program, equivalence of the nets, verification, semantic relations.*

DOI: 10.21667/1995-4565-2019-70-15-26

References

1. **Kulyamin V. V.** Metody verifikacii programmnogo obespecheniya. *Vserossijskij konkursnyj otbor obzorno-analiticheskih statej po prioritetnomu napravleniyu «Informacionno-telekommunikacionnye sistemy»*. 2008, vol. 117. (in Russian).

2. **Ivutin A. N., Troshina A. G., Esikov D. O.** Primenenie semanticheskih setej Petri-Markova dlya resheniya zadachi rasparallelivaniya algoritmov. *Vestnik Ryazanskogo gosudarstvennogo radiotekhnicheskogo universiteta*. 2016, no. 58, pp. 49-56. (in Russian).

3. **Legalov A. I.** Funkcional'nyj yazyk dlya sozdaniya arhitekturno-nezavisimyh parallel'nyh program. *Vychislitel'nye tekhnologii.* 2005, vol. 10, no. 1 (in Russian).

4. **Kropacheva M. S., Legalov A. I.** Formal'naya verifikaciya programm, napisannyh na funkcional'nopotokovom yazyke parallel'nogo programmirovaniya. *Modelirovanie i analiz informacionnyh sistem*. 2015, vol. 19, no. 5, pp. 81-99. (in Russian).

5. **Siegel S. F., Zirkel T. K.** Automatic formal verification of MPI-based parallel programs. *ACM Sigplan Notices*. 2011, vol. 46, no. 8, pp. 309-310.

6. **Afanas'ev K. E., Vlasenko A. Yu.** Semanticheskie oshibki v parallel'nyh programmah dlya sistem s raspredelennoj pamyat'yu i metody ih obnaruzheniya sovremennymi sredstvami otladki. *Vestnik Kemerovskogo gosudarstvennogo universiteta*. 2009, no. 2. (in Russian).

7. **Kulyamin V. V.** Integraciya metodov verifikacii programmnyh system. *Programmirovanie*. 2009, vol. 35, no. 4, pp. 41-55. (in Russian).

8. **Hopkroft D. E., Motvani R., Ul'man D.** *Vvedenie v teoriyu avtomatov, yazykov i vychislenij*. 2008. (in Russian).

9. **Simon G. A.** An extended finite state machine approach to protocol specification. *Proc. of 2-nd International Workshop on Protocol Specification. Testing and Verification*, pp. 113-133, North-Holland Publishing Co., 1982.

10. **Cheng K. T., Krishnakumar A. S.** Automatic functional test generation using the extended finite state machine model. *30th ACM/IEEE Design Automation Conference. IEEE*, 1993, pp. 86-91.

11. **Luo G., von Bochmann G., Petrenko A.** Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method // IEEE Transactions on Software Engineering. 1994, vol. 20, no. 2, pp. 149-162.

12. **Balarin F. et al.** Formal verification of embedded systems based on CFSM networks. *33rd Design Automation Conference Proceeding. IEEE*, 1996, pp. 568-571.

13. **Gurevich Y.** Evolving Algebras: An Introductory Tutorial. *Bull. EATCS*. 1991, vol. 43, pp. 264-284.

14. **Gurevich Y.** Posledovatel'nye mashiny abstraktnyh sostoyanij ohvatyvayut posledovatel'nye algoritmy. *Sistemnaya informatika*. 2004, vol. 9, pp. 7-50. (in Russian).

15. **Piterson Dzh.** *Teoriya setej Petri i modelirovanie sistem*. M.: Mir, 1984. 264 p. (in Russian).

16. **Ramchandani C.** Analysis of asynchronous concurrent systems by timed Petri nets. *PhD Thesis. Cambridge*, Mass.: MIT, Dept. Electrical Engineering, 1974.

17. **Kösters G., Six H. W., Winter M.** Coupling use cases and class models as a means for validation and verification of requirements specifications. *Requirements engineering*. 2001, vol. 6, no. 1, pp. 3-17.

18. **Hoare C. A. R.** An axiomatic basis for computer programmingю *Communications of the ACM*. 1969, vol. 12, no. 10, pp. 576-580.

19. **Harel D., Kozen D., Tiuryn J.** Dynamic logic. *Handbook of philosophical logic*. Springer, Dordrecht, 2001, pp. 99-217.

20. **Burdonov I. B. i dr.** Obzor podhodov k verifikacii raspredelennyh system. *ISP RAN*, preprint. 2003, vol. 16. (in Russian).

21. **Grumberg O., Long D. E.** Model checking and modular verification. *ACM Transactions on Programming Languages and Systems* (TOPLAS). 1994, vol. 16, no. 3, pp. 843-871.

22. **Henzinger T. A., Jhala R., Majumdar R.** The BLAST software verification system. *International SPIN Workshop on Model Checking of Software*. Springer, Berlin, Heidelberg, 2005, pp. 25-26.

23. **Karpov Yu. G.** Model shecking. *Verifikaciya parallel'nyh i raspredelennyh programmnyh sistem*. BHV-Peterburg, 2010. (in Russian).

24. **Lattner C.** LLVM Language Reference Manual: LLVM compiler infrastructure. *LLVM Project*. 2003-2017. URL: http://llvm.org/docs/index.html (дата обращения: 30.08.2019).

25. **Ivutin A., Troshina A., Novikov A. S.** Low-level Code Auto-tuning for State-of-the-art Multicore Architectures. *29th International Conference Radioelektronika (RADIOELEKTRONIKA). IEEE*, 2019, pp. 1-6.

26. **Ivutin A. N., Troshina A. G.** Semantic Petri-Markov nets for automotive algorithms transformations. *28th International Conference Radioelektronika (RADIOELEKTRONIKA). IEEE*, 2018, pp. 1-6.

27. **Uil'yams E.** Parallel'noe programmirovanie na C++ v dejstvii. Praktika razrabotki mnogopotochnyh programm. Per. s anglyu Slinkin A. V. M.: DMK Press, 2012. 672 p. (in Russian).

28. **Andrianov P. S., Mutilin V. S., Horoshilov A. V.** Metod legkovesnogo staticheskogo analiza dlya poiska sostoyanij gonok. *Trudy Instituta sistemnogo programmirovaniya RAN*. 2015, vol. 27, no. 5. (in Russian).

29. **Prokopenko A. S.** *Staticheskij analiz uslovij gonki v parallel'nyh programmah na razdelyaemoj pamyati*: dis. Moskva, 2010, 107 p. (in Russian).

30. **Trifanov V. Yu., Citelov D. I.** Dinamicheskie sredstva obnaruzheniya gonok v parallel'nyh programmah. *Komp'yuternye instrumenty v obrazovanii*. 2011, no. 6 (in Russian).

31. **McConnell S.** *Sovershennyj kod. Master-klass*. Per. s angl. M.: Izdatel'stvo «Russkaya redakciya», 2010. 896 p. (in Russian).