ЧЕБЫШЕВСКИЙ СБОРНИК
Том 19. Выпуск 2

УДК 519.[172-178] DOI 10.22405/2226-8383-2018-19-2-151-162

О некоторых комбинаторных свойствах деревьев процессов LINUX

Ефанов Николай Николаевич — аспирант кафедры информатики и вычислительной математики МФТИ, преподаватель, программист ЛЦССН МФТИ.
e-mail: nefanov90@gmail.com

Аннотация

В работе рассматривается структура данных - дерево процессов Linux, возникающая вследствие иерархической схемы порождения процессов в Unix-подобных операционных системах. Целью исследования является выделение свойств деревьев процессов Linux, позволяющие заключить о применимых методах анализа таких деревьев, с целью решения задачи сохранения и восстановления состояний исполняемых сред операционной системы Unix-подобных операционных систем.

Формулируется обратная дискретная задача восстановления цепочек системных вызовов, порождающих некоторое дерево процессов, а также ряд ограничений на вид системного вызова и утверждение о существовании решения, заключающее корректность ввода. Приводятся комбинаторные оценки общего количества деревьев при порождении системным вызовом fork, вводится поправка на различимость идентификаторов.

Осуществляется обоснование возможности индексирования деревьев по узлам, благодаря образованию некорневыми идентификаторами симметрической группы. Таким образом, доказывается функциональная эквивалентность автоморфных деревьев с перестановками некорневых идентификаторов. Демонстрируется комбинаторный взрыв числа функционально различных деревьев при добавлении нового системного вызова. Ввиду приведённых оценок, проводится заключение о неэффективности восстановления деревьев процессов перебором или прямым поиском, предлагается идея построения некоторых восстанавливающих математических формализмов, учитывающих структуру задачи.

Далее рассматривается свойство наследования атрибутов в дереве процессов, позволяющее локализовать область поиска нужного атрибута при проверке применимости правила системного вызова, таким образом снизив количество проверок. Заключается свойство сегментации дерева процессов Linux. На базе приведённых свойств формулируется заключение о целесообразности построения решения задачи восстановления цепочек системных вызовов, восстанавливающих дерево процессов, на базе теории формальных языков и грамматик, используя формализмы класса мягко-контекстно-зависимых. Приводятся обзорно альтернативные способы решения задачи.

Ключевые слова: математическое моделирование, перечислительная комбинаторика, группы, деревья, Unix-подобные операционные системы, системные вызовы, дерево процессов, восстановление по контрольным точкам, формальные грамматики.

Библиография: 22 названий.

Для цитирования:
Н. Н. Ефанов. О некоторых комбинаторных свойствах деревьев процессов LINUX // Чебышевский сборник, 2018, т. 19, вып. 2, с. 151–162.

CHEBYSHEVSKII SBORNIK
Vol. 19. No. 2

UDC 519.[172-178] DOI 10.22405/2226-8383-2018-19-2-151-162

On some combinatorial properties of LINUX process trees

Efanov Nikolai Nikolayevich — graduate student of the department of informatics and computational mathematics of MIPT, teacher, programmer of the LLSSN MIPT.
e-mail: nefanov90@gmail.com

Abstract

The paper examines the Linux process tree data structure, which arises from the hierarchical scheme of processes generation in Unix-like operating systems. The purpose of study is to highlight the properties of the Linux process trees, which allow to conclude the applicable methods for analyzing such trees, in aim to solve the checkpoint-restore problem of executable environments in Unix-like operating systems.

The inverse discrete problem of restoring chains of system calls that generate a certain tree of processes is formulated, as well as a number of restrictions on the form of the system call and an assertion about the existence of a solution that concludes the correctness of the input.

Combinatorial estimation of total trees number, which are provided by fork system call, is presented, and correction is noted for the discernibility of identifiers. The feasibility of indexing by nodes is substantiated, due to the formation of non-root identifiers of the symmetric group. Thus, the functional equivalence of automorphic trees with permutations of non-root identifiers is proved. A combinatorial explosion of the functionally different trees number is shown by the procedure of adding a new system call. In view of the above estimations, a conclusion about the ineffectiveness of process trees restoring by bruteforce or direct search is drawn. The idea of constructing restoring mathematical formalisms that take into account the structure of the problem is proposed.

Next, the inheritance property of attributes in the process trees is examined, which allows to localize a required attribute when checking the applicability of a system call rule, thus reducing the number of checks. The segmentation property of the Linux process trees is provided. On the basis of the above properties, the conclusion is formulated on the goal of constructing a solution of restoring the syscall chains, which constructing a certain process tree, on the basis of the theory of formal languages and grammars, using formalisms of the class of mildly-context-sensitive. The alternative methods of solution are reviewed too.

Keywords: mathematical modeling, enumeration combinatorics, groups, trees, Unix-like operating systems, system calls, process tree, checkpoint restore, formal grammars.

Bibliography: 22 titles.

For citation:
N. N. Efanov, 2018, "On some combinatorial properties of LINUX process trees" , *Chebyshevskii sbornik*, vol. 19, no. 2, pp. 151–162.

1. Введение

Структуры данных, возникающие в различных компьютерных системах, позволяют разрабатывать математические модели из прямых принципов функционирования данных систем. Одной из таких задач применительно к операционным системам (ОС) является восстановление состояния среды исполнения – процесса или группы процессов, контейнера, виртуальной машины, с целью поддержки функции сохранения-восстановления по контрольным точкам, а также живой миграции [1, 2, 3, 4]. Особенность создания нового процесса в Unix-подобных ОС наследованием от другого процесса через системный вызов fork() порождает целевую структуру, рассматриваемую в данной работе – дерево процессов Linux, состоящее из вершин, содержащих описание процессов как набор атрибутов – идентификаторов процесса, сессии, группы процессов, пользователя, открытых файловых дескрипторов и других ресурсов, используемых программой, исполняемой в контексте данного процесса, и рёбер, описывающих иерархические связи между процессами (Рис. 1). Базовым способом изменить некоторый ат-

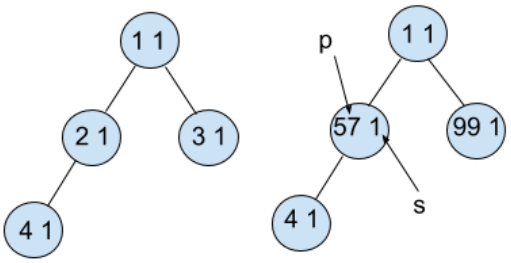

Рис. 1: Два функционально эквивалентных дерева процессов. Идентификаторы $< 1 >, < 2 >$ – идентификаторы процесса P и сессии S соответственно. Корневой процесс всегда имеет идентификаторы “1,1”.

рибут процесса в современных ОС является системный вызов – передача управления в ядро, исполняемое в привилегированном режиме. Целью работы является выявление некоторых свойств деревьев процессов Linux из перечислительной комбинаторики, теории графов, заключение о возможных методах решения задачи восстановления цепочек системных вызовов, приводящих к некоторому целевому дереву процессов, то есть фактически восстанавливающих некоторую среду исполнения операционной системы Linux.

2. Задача восстановления цепочек системных вызовов

Прежде чем перейти к непосредственной оценке числа деревьев и выводам по подходящим методам восстановления, следует кратко сформулировать прикладную задачу, опираясь на работы [2, 3]: разработать алгоритм, получающий на вход дерево процессов, описывающее конфигурацию исполняемой среды операционной системы, возвращающий цепочки системных вызовов для восстановления входного дерева из некоторого начального состояния. В работах [2, 3] отмечено, что цепочки системных вызовов можно хранить в дереве, содержащем вершины-описания процессов в некотором состоянии, и рёбра с метками-системными вызовами, учётом того, что:

- Рассматриваемые системные вызовы, не изменяющие количество вершин, напрямую изменяют атрибуты лишь вызвавшего процесса, к примеру, вызов setpgid(pid, pgid) ограничен до setpgid(0, pgid). Таким образом, получить нужную цепочку после восстановления можно прохождением по любой ветви дерева от корня к листовым вершинам [2, 3].

- Системный вызов fork, порождающий потомки у вызывающего процесса, восстанавливается приписыванием исходным рёбрам соответствующей метки, без добавления нового ребра.
- Системный вызов exit, завершающий процесс и переупорядочивающий процессы-потомки к корню, не рассматривается в анализе. Предполагается, что восстановление вершин, соответствующее завершенным процессам, с последующим обратным присоединением, осуществляется некоторым набором эвристических правил [2, 3] поиска либо восстановления соответствующей сессии.

С учётом вышеописанных ограничений, задача восстановления дерева процессов цепочками системных вызовов может быть сформулирована так:

Получая на вход дерево процессов $D = \{V, E\}$, где V – множество вершин, E – множество рёбер, возможно, с меткой 'exit', получить дерево $T = \{V^+, E^+\}$, где $V^+ = V \cup V_h$, V_h – множество добавленных вершин, описывающих промежуточные состояния процессов, E^+ – множество рёбер с метками-системными вызовами из множества R, выполнение которых восстанавливает D из стартового состояния I[1]. При этом:

$$E \cap E^+ \subset E^+ \tag{1}$$

$$V = V \cap V^+ \tag{2}$$

Условие (1) заключает то, что некоторые смежные вершины во входном дереве могут не являться смежными в выходном (Рис. 2). Условие (2) заключает неукорачивание дерева, что соответствует отдельному рассмотрению вызова exit [2, 3], при этом у некоторых рёбер на входе уже может быть метка 'exit'.

Утверждение 1. *Пусть S – множество различных конфигураций исполняемых сред ОС. Тогда $\forall\ D \in S\ \ \exists T : T$ – решение задачи.*

Обоснование: допустим, что $\exists D \in S : \forall T$ не является решением. Тогда не существует последовательностей системных вызовов, восстанавливающих D, то есть $D \notin S$. **Противоречие.**

С учётом вышесказанного, наиболее тривиальным подходом к восстановлению является прямое порождение нужного дерева процессов некоторым переборным алгоритмом. Целью данной работы является демонстрация сложности такого решения, ввиду комбинаторных оценок количества деревьев.

3. Оценки числа деревьев

Данный раздел приводит оценки числа деревьев процессов из перечислительной комбинаторики. Ключевыми параметрами в таких оценках является число вершин, а также число атрибутов и множество принимаемых ими значений, связанные с поддержкой системных вызовов.

Продемонстрируем оценку для количества различных деревьев, поддерживающих системный вызов fork, приводимую в работе [2]:

$$F(n) = \sum_{i=2}^{n-1} i^{i-2}, \tag{3}$$

[1] Стартовым состоянием, не теряя общности, можно считать корневой Init-процесс, имеющий единичные значения идентификатора, сессии, группы и т.д.

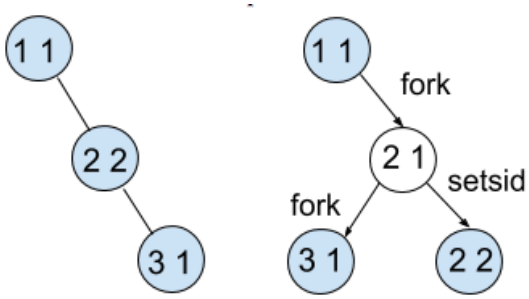

Рис. 2: Входное дерево процессов T с поддержкой сессий и полученное дерево D с метками-системными вызовами на рёбрах. Вершина “2 1” – восстановленное промежуточное состояние. Вершина “3 1” присоединена к состоянию процесса “2” до выполнения setsid.

где $n = 2^{16}$ – максимально возможное число процессов в ОС Linux.

Данный результат в точности соответствует суммированию количества различных остовных деревьев на i вершинах по формуле Кейли [5], с учётом выделенной вершины для корневого процесса, связанного с ядерным с идентификатором 0 [6, 7]. Суммирование проводится от 2 до $n-1$ ввиду существования одной конфигурации для пары корневой-ядерный. Индуктивное доказательство (3) приводится в работе [2], без учёта значений идентификатора процесса. Тем не менее, простой переборный алгоритм, либо алгоритм, требующий строгого соответствия сохраняемого и восстанавливаемого идентификатора, может существенно опираться именно на его значение. Требование к рассмотрению идентификаторов процесса вносит существенную поправку в приведённую оценку.

Утверждение 2. *Количество генерируемых деревьев процессов с учётом различимости по идентификаторам равняется:*

$$F(n) = \sum_{i=2}^{n-1} \binom{2^{16}-2}{(i-1)} (i-1)! i^{i-2} \tag{4}$$

Доказательство. Зафиксируем количество процессов i. Пусть $p = 2 \dots 2^{16} - 1$ – идентификатор некорневого процесса, таким образом, выбор одного идентификатора можно совершить $(2^{16} - 2)$-способами. Пусть P – множество идентификаторов в вершинах, без учёта корневой, $P^{'}$ – некоторая перестановка элементов P. Число способов назначить множество значений элементов $P^{'}$ $(i-1)$-процессам равно числу размещений $(i-1)$ в $(2^{16} - 2)$:

$$\binom{2^{16}-2}{(i-1)} (i-1)! \tag{5}$$

в точности равному поправке (4). **Что и требовалось доказать.**

Следует отметить, что в большинстве практических приложений строгого восстановления значения идентификатора не требуется: различные по атрибутам деревья могут быть функционально эквивалентными, то есть описывать идентичные состояния исполняемой среды (Рис. 1). Приведём одно практически важное групповое свойство деревьев процессов.

Утверждение 3. *Идентификаторы некорневых процессов образуют симметрическую группу S_{n-1}, при этом выполняется функциональная эквивалентность тривиально автоморфных деревьев [8], вершины которых пронумерованы перестановками из таких идентификаторов.*

Обоснование: применим строчную нотацию для дерева процессов, представленную в работах [2, 3]: $p\ g\ s\ [\ children\ ]$, где p, g, s – идентификаторы процесса, группы, сессии некоторого

процесса, [*children*] – список, возможно пустой, прямых потомков процесса, задаваемых в этой же нотации, описывающей таким образом всё дерево рекурсивно. В такой нотации fork имеет вид: $* \ * \ * \ [*] \longrightarrow * \ * \ * \ [* \setminus 1 \ /2 \ /3, \ /4]$, где $*$ – метка, принимающая любое значение из возможных для идентификатора, $* \setminus a$ – операция исключения значения a из возможных, $/b$ – подстановка значения идентификатора из b-й метки. Таким образом, нет ограничений на возможные идентификаторы, кроме исключения дубликатов и фиксированного значения "1" для корневого процесса, то есть любые перестановки некорневых идентификаторов на некотором дереве описывают функционально эквивалентные деревья. Эквивалентность деревьев при любых некорневых перестановках идентификаторов говорит об образовании симметрической группы S_{n-1} такими идентификаторами, по построению. **Что и требовалось продемонстрировать.**

Утверждение (3) предоставляет обоснование для нормализации идентификаторов деревьев процессов: пронумеровав вершины по некоторой единой процедуре, можно перейти к рассмотрению лишь функционально различных деревьев, количество которых, очевидно, равняется (3).

Следующим свойством, упрощающим восстановление с поддержкой fork является фактическое отражение данного вызова в структуре дерева: вершина-потомок порождается с использованием как минимум одного fork [2, 3]. Несмотря на большое количество различных деревьев (2), фактически, восстановление цепочек fork можно провести обходом дерева за линейное время [9, 10]. Большинство других системных вызовов не отражены в структуре дерева, что затрудняет их восстановление.

Работа [2] приводит оценку числа деревьев с учётом сессий и системного вызова setsid с целью продемонстрировать быстрый рост количества различных деревьев при добавлении атрибута сессии, либо наследуемого от родителя к потомку, либо выставляемого локально для некоторого процесса, с образованием новой сессии (Рис. 2). Приведём более сильное утверждение, которое годится для более широкого класса системных вызовов: выставление нового значения некоторого атрибута, либо наследование, либо выставление чужого значения атрибута.

Утверждение 4. *Пусть $F(n)$ – число деревьев из n процессов с поддержкой некоторых системных вызовов. Рассмотрим новый вызов k_ call(k), изменяющий атрибут вызывающего процесса k, наследуемый при fork: k_ call(k') = k', k_ call(0) = pid, где pid – идентификатор вызвавшего процесса. Тогда поправка для числа деревьев процессов:*

$$F(n, k\_call) = F(n) \sum_{m=0}^{n-1} \binom{n}{m} (m+1)^{n-m-1} \tag{6}$$

Доказательство. Пусть в дереве из n процессов m некорневых совершили вызов k_call(0), тогда $\exists \ z(n,m) = \binom{n}{m}$ таких конфигураций. Каждый из оставшихся $n-m-1$ некорневых процессов может либо сохранить наследованное значение для k, либо установить его m способами (Рис. 3), то есть $\exists \ y(n,m) = (m+1)^{n-m-1}$ их конфигураций. Суммируя количество различных конфигураций по $m = 0 \ldots n-1$, получим:

$$\sum_{m=0}^{n-1} z(m) \, y(n,m) \tag{7}$$

Выражение (7) в точности равно поправке из оценки (6), **что и требовалось доказать.**

Рассмотрим 1-й член суммы (7) s_0. Тогда $m = 0$, $s_0 = 2^{n-1}$. Данное равенство демонстрирует комбинаторный взрыв [11] при добавлении новых системных вызовов: число деревьев процессов возрастает более чем в полином раз, и степень полинома $n - 1 \approx n$ при достаточно

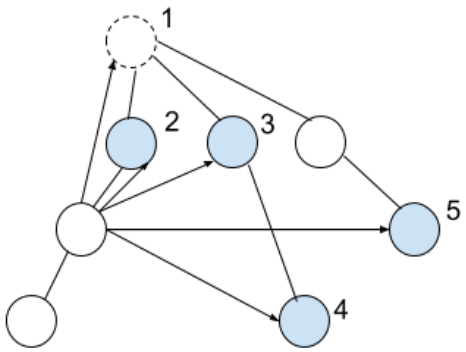

Рис. 3: К доказательству утверждения 4. Вершины с метками – процессы, совершившие вызов k_call(0), остальные вершины либо наследуют, либо изменяют значение атрибута на некоторое значение из дерева, что показано стрелками. Значение “1” также может быть наследовано.

больших n ограничена только числом вершин дерева. Данный результат демонстрирует практическую трудность применения прямых методов перебора и поиска подходящих деревьев T ввиду экспоненциального роста числа деревьев, восстановление системных вызовов для которых не следует напрямую из вида дерева, как при поддержке fork. Данный факт мотивирует поиск более строгих методов восстановления, краткому обзору которых посвящён раздел 5.

4. Группировка процессов и заключающие правила

В поддержку построения указанных выше способов восстановления деревьев, рассмотрим свойство группировки процессов в сессии: по умолчанию идентификатор s наследуется процессом от родителя, сохраняя сессию, либо порождается новая (Рис. 4).

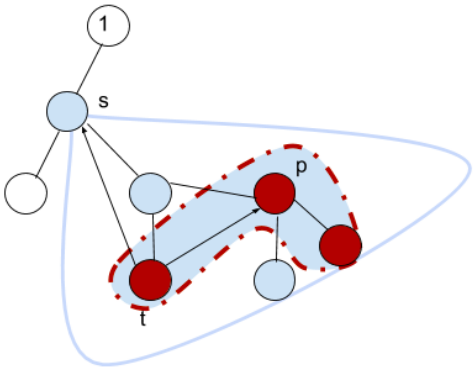

Рис. 4: Проверка некоторого наследуемого атрибута процесса p из процесса t. p и t лежат в одной сессии с лидером s. При этом достаточно обойти поддерево от s.

Следовательно, при восстановлении цепочек системных вызовов требуется учитывать, порождён ли некоторый процесс от родительского состояния до или после выполнения setsid. Простейшие правила проверки заключаются в восходящем поиске идентификатора сессии ребёнка по текущей ветви дерева: $v_t.s \in V_{above}.s$, где $v_t.s$ – идентификатор сессии проверяемого процесса, $V_{above}.s$ – множество идентификаторов сессий из вершин по ветви вверх от родительского до корневого процесса. Подобные операции сравнения, как и свойства группировки процессов по атрибутам – группам процессов и пользователей, позволяют не более чем за $O(n)$ по времени проверить некоторые k атрибутов в дереве, $k \ll n$, то есть для n-вершин можно

совершить проверки за в худшем случае за $O(n^2)$ обходом дерева в глубину или в ширину [9, 10].

Ввиду того, что многие системные вызовы изменяют свойства процессов только в рамках некоторой сессии, оценка $O(n^2)$ на практике может быть меньше: для таких вызовов нет необходимости обходить всё дерево, достаточно найти лидера сессии и осуществить поиск в его поддереве. Другим улучшением является индексирование дерева и быстрый поиск среди вершин который может улучшить проверки от $O(1)$ до $O(n)$-раз, в зависимости от использованного алгоритма [10].

5. О формальных грамматиках, анализе деревьев и других методах решения

Данный раздел кратко приводит некоторые математические методы решения, несмотря на то, что в современных компьютерных системах наибольшее распространение получили техники профилирования и восстановления наборами эвристик. С обзором данных решений можно ознакомиться в обзорах к работам [3, 4]. Вообще говоря, математическая формализация данной задачи является относительно новым вопросом, продиктованным активным развитием высоконагруженных и распределённых систем. Одним из подходов, предложенным и развиваемым автором, является построение формальных грамматик системных вызовов.

В такой постановке производится разбор дерева D как строки на некотором языке, порождённом формальной грамматикой $G = \{A, N, P, < s >\}$, где: A – множество терминалов – конечные состояния (процессов), N – множество промежуточных состояний, P – набор правил системных вызовов и порождения некоторых вспомогательных конструкций, $< s > \in N$ – стартовый нетерминал. Выходом является дерево разбора T с метками на рёбрах – системными вызовами, приводящими к данному D, что соответствует постановке задачи. Очевидно, данная грамматика является существенно неоднозначной: существует несколько способов породить эквивалентные деревья процессов, однако это не противоречит постановке задачи: требуется получить произвольное дерево T из множества деревьев, приводящих к D в условиях (1)-(2).

Получение оптимальных цепочек системных вызовов также является открытым вопросом для исследования. На данный момент разработаны методы восстановления на базе разбора строчной нотации дерева проверкой атрибутов двухпроходным анализатором [12] с механизмом вложенного стекового кадра, поддерживающим системные вызовы fork, setsid, setpgid и exit [2, 3]. Очевидно, разбор строк обычными грамматиками [13] с проверками атрибутов плохо применим к формулируемой задаче: зависимости атрибутов одних процессов от других и относительно низкая выразительность дизъюнктивных грамматик мотивируют поиск подходящих мягко-контекстно-зависимых формализмов [14, 15, 16, 17], учитывающих топологию входных данных как дерева, и поддерживающих контекст некоторого вида [15, 16, 17, 18]. На данный момент автором проводятся работы по построению двухкомпонентной параметризованной грамматики вставки деревьев [18, 19], а также эквивалентной ей по выразительной способности грамматики оборачивания пар [13] с поддержкой ветвлений.

Из других способов решения задачи следует отметить работу [20], являющуюся попыткой применения матричного анализа к поставленной задаче.

Из работ, близких по виду рассматриваемой проблемы, но не имеющих прямого отношения к ней, следует выделить исследование [21], посвящённое частному случаю анализа деревьев процессов и порождающих системных вызовов, из которого можно заимствовать результаты на случай поддержки параллелизма, и [22], представляющую применение атрибутных грамматик к некоторым вычислительным задачам.

6. Заключение

Несмотря на полученные нелинейные оценки роста числа деревьев при добавлении вершин и комбинаторный взрыв при добавлении системных вызовов, симметрическая группа идентификаторов и наследование атрибутов заключают существование эффективных строгих методов решения задачи в условиях (1)-(2), а также отдельного эвристического восстановления завершенных процессов [3]. Тем не менее, ряд вопросов также является открытым: оптимальность цепочек системных вызовов, метрики эффективности, статистические свойства деревьев процессов и применимость приближённых методов восстановления. Дальнейшие исследования автора будут посвящены построению методов решения, указанных в разделе 5, параллельно с исследованиями открытых вопросов.

СПИСОК ЦИТИРОВАННОЙ ЛИТЕРАТУРЫ

1. Mirkin A., Kuznetsov A., Kolyshkin K. Containers checkpointing and live migration // Proceedings of the In Ottawa Linux Symposium, vol. 2, pp. 85-90 (2008).

2. Ефанов Н.Н., Емельянов П. В. Построение формальной грамматики системных вызовов // М. Информационное обеспечение математических моделей, 2017, 83-91 сс.

3. Efanov N. N., Emelyanov P. V. Constructing the formal grammar of system calls // In Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17). ACM, New York, NY, USA, 2017, Article 12, 5 pages. doi: 10.1145/3166094.3166106

4. Bozyigit M., Wasiq M. User-level process checkpoint and restore for migration // ACM SIGOPS Operating Systems Review, 2017, vol. 35 no. 2, pp. 86-96. doi: 10.1145/377069.377091

5. Cayley A. A theorem on trees. // Collected Mathematical Papers, 1897, vol. 13. pp. 26-28.

6. Clemente Izurieta, James Bieman. The evolution of FreeBSD and linux. // In Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering (ISESE '06). ACM, New York, NY, USA, 2006, pp. 204-211. doi: 10.1145/1159733.1159765

7. Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. Coevolution of variability models and related artifacts: a case study from the Linux kernel // In Proceedings of the 17th International Software Product Line Conference (SPLC '13). ACM, New York, NY, USA, 2013, pp. 91-100. doi: 10.1145/2491627.2491628

8. Белоусов А. И. Дискретная математика, 4-е изд., М.: МГТУ имени Н. Э. Баумана, 2006, 744 с.

9. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, 2nd Edition, MIT Press and McGraw-Hill , 2001. ISBN 0-262-03293-7.

10. Knuth, Donald E., The Art of Computer Programming Vol 1. 3rd edition, Boston: Addison-Wesley, 1997. ISBN 0-201-89683-4

11. Krippendorff, Klaus. "Combinatorial Explosion"// Web Dictionary of Cybernetics and Systems. PRINCIPIA CYBERNETICA WEB. Retrieved 29 November 2010. URL: `http://pespmc1.vub.ac.be/ASC/Combin_explo.html` (Дата обращения: 31.05.2018).

12. Alfred V. Aho , Jeffrey D. Ullman , The theory of parsing, translation, and compiling, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972. URL: dl.acm.org/citation.cfm?id=578789 (Дата обращения: 31.05.2018).

13. Охотин А. Формальные грамматики и вычислительная сложность синтаксического анализа. СПбГУ, программа "Математика", 12 ноября 2017 г.

    URL: https://compsciclub.ru/media/slides/for\-mal_grammars_2017_autumn/2017_11_12_formal_grammars_2017_autumn.pdf (Дата обращения: 31.05.2018).

14. David J. Weir. Characterizing Mildly Context-Sensitive Grammar Formalisms. Ph.D. thesis, University of Pennsylvania, Philadelphia, USA, 1988. doi:10.1145/3166094.3166106

15. M. A. Alonso and V. J. Díaz. Variants of mixed parsing of TAG and TIG // Traitement Automatique des Langues (TAL), 2003, no. 44(3), pp. 41–65.

16. Barash, Mihail. Defining Contexts in Context-Free Grammars, Turku Center for Computer Science, TCCS Dissertation no. 204, 2015. URL: https://www.doria.fi/bitstream/handle/10024/113793/TUCSDissertationD204.pdf (Дата обращения: 31.05.2018).

17. Boullier, P. On Multicomponent TAG parsing. // In 6th conference annuelle sur le Traitement Automatique des Langues Naturelles (TALN 1999), Cargse, Corse, France, 1999, pp. 321–326.

18. Boullier P., Sagot B. Multi-Component Tree Insertion Grammars. // In: de Groote P., Egg M., Kallmeyer L. (eds) Formal Grammar. FG 2009. Lecture Notes in Computer Science, vol 5591. Springer, Berlin, Heidelberg, 2011.

19. Claire Gardent and Laura Kallmeyer. Semantic construction in feature-based TAG. // In Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1 (EACL '03), vol. 1, Association for Computational Linguistics, Stroudsburg, PA, USA, 2003, pp. 123-130. doi: 10.3115/1067807.1067825

20. Marina Kudinova and Pavel Emelyanov. Building Mathematical Model for Restoring Processes Tree during Container Live Migration. // IVth International Conference on Engineering and Telecommunication (EnT), November 2017, Dolgoprudniy. doi: 10.1109/ICEnT.2017.41

21. Zengin M., Vafeiadis V. A programming language approach to fault tolerance for fork-join parallelism // Proceedings of the 7th International Symposium on Theoretical Aspects of Software Engineering (TASE 2013), Max Planck Institute for Software Systems (MPI-SWS), Saarsbruchen, Germany, 2013.

22. Adel Cherif, Masato Suzuki, and Takuya Katayama. A Replication Technique Based on a Functional and Attribute Grammar Computation Model. In Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96), IEEE Computer Society, Washington, DC, USA, 1996, p. 266.

REFERENCES

1. Mirkin A., Kuznetsov A., Kolyshkin K, 2008, "Containers checkpointing and live migration", Proceedings of the In Ottawa Linux Symposium. Ottawa, Ontario, Canada, vol.2, pp. 85-90.

2. Efanov N.N., Emelyanov P.V. 2017, "Postroenie formal'noj grammatiki sistemnyh vyzovov" , Informacionnoe obespechenie matematicheskih modelej, Moscow, Russia, pp. 83-91.

3. Efanov N. N., Emelyanov P. V. “Constructing the formal grammar of system calls”, In Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR ’17), ACM, New York, NY, USA, Article 12, 5 pages. doi: 10.1145/3166094.3166106

4. Bozyigit M., Wasiq M., 2001, “User-level process checkpoint and restore for migration”, ACM SIGOPS Operating Systems Review, vol. 35, no. 2, p.86-96. doi: 10.1145/377069.377091

5. Cayley A. 1897, “A theorem on trees”, Collected Mathematical Papers, Cambridge University Press, vol. 13, pp. 26-28.

6. Clemente Izurieta and James Bieman. 2006, “The evolution of FreeBSD and linux”, In Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering (ISESE ’06), ACM, New York, NY, USA, 204-211. doi: 10.1145/1159733.1159765

7. Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba 2013, “Coevolution of variability models and related artifacts: a case study from the Linux kernel”, In Proceedings of the 17th International Software Product Line Conference (SPLC ’13), ACM, New York, NY, USA, 91-100. doi: 10.1145/2491627.2491628

8. Belousov, A.I. 2006, Discretnaya matematika, 4th edition, Bauman University (BMSTU), Moscow, p. 349.

9. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001, Introduction to Algorithms, 2nd Edition, MIT Press and McGraw-Hill. ISBN 0-262-03293-7.

10. Knuth, Donald E. 1997, The Art of Computer Programming, Vol 1., 3rd edition, Boston: Addison-Wesley, ISBN 0-201-89683-4

11. Krippendorff, Klaus, 2010, "Combinatorial Explosion Web Dictionary of Cybernetics and Systems, PRINCIPIA CYBERNETICA WEB, Retrieved 29 November 2010. Available at: `http://pespmc1.vub.ac.be/ASC/Combin_explo.html`

12. Alfred V. Aho , Jeffrey D. Ullman 1972, The theory of parsing, translation, and compiling, Prentice-Hall, Inc., Upper Saddle River, NJ, USA. Available at: `dl.acm.org/citation.cfm?id=578789`

13. Okhotin A, 2017, “Formal’nye grammatiki i vychislitelnaya slojnost’ sintaksicheskogo analiza”, “Matematika” educational program, SPbSU. Available at: `https://compsciclub.ru/media/slides/formal_grammars_2017_autumn/2017_11_12_formal_grammars_2017_autumn.pdf`

14. David J. Weir. 1988, Characterizing Mildly Context-Sensitive Grammar Formalisms, Ph.D. thesis, University of Pennsylvania, Philadelphia, USA. doi: 10.1145/3166094.3166106

15. Alonso M. A. and Díaz V. J. 2003, “Variants of mixed parsing of TAG and TIG”, Traitement Automatique des Langues (T.A.L.), no. 44(3), pp. 41–65.

16. Barash Mihail, 2015, Defining Contexts in Context-Free Grammars, Turku Center for Computer Science, TCCS Dissertation No 204. Available at: `https://www.doria.fi/bitstream/handle/10024/113793/TUCSDissertationD204.pdf`

17. Boullier P. 1999, “On Multicomponent TAG parsing”, In 6th conference annuelle sur le Traitement Automatique des Langues Naturelles (TALN 1999), Cargse, Corse, France, pp. 321–326.

18. Boullier P., Sagot B. 2011. “Multi-Component Tree Insertion Grammars”, In: de Groote P., Egg M., Kallmeyer L. (eds) Formal Grammar, Lecture Notes in Computer Science, vol. 5591, Springer, Berlin-Heidelberg.

19. Claire Gardent and Laura Kallmeyer. 2003, “Semantic construction in feature-based TAG”, In Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1 (EACL ’03), Vol. 1, Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 123-130. doi: 10.3115/1067807.1067825

20. Marina Kudinova and Pavel Emelyanov. 2017, “Building Mathematical Model for Restoring Processes Tree during Container Live Migration”, IVth International Conference on Engineering and Telecommunication (EnT), November 2017, Dolgoprudniy. doi: 10.1109/ICEnT.2017.41

21. Zengin M, Vafeiadis V. 2013, “A programming language approach to fault tolerance for fork-join parallelism”, Proceedings of the 7th International Symposium on Theoretical Aspects of Software Engineering (TASE 2013), Max Planck Institute for Software Systems (MPI-SWS), Saarsbruchen, Germany, 2013. Available at: `http://plv.mpi-sws.org/ftpar/tase2013-ftpar.pdf`

22. Adel Cherif, Masato Suzuki, and Takuya Katayama. 1996, “A Replication Technique Based on a Functional and Attribute Grammar Computation Model”, In Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE ’96), IEEE Computer Society, Washington, DC, USA, p. 266.

Получено 13.06.2018
Принято в печать 17.08.2018