MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 4, 2024
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

An Exact Schedulability Test for Real-Time Systems With Abstract Scheduler on Multiprocessor Platforms

N. O. Garanina[1]

DOI: 10.18255/1818-1015-2024-4-474-494

[1]Institute of Automation and Electrometry SB RAS, Novosibirsk, Russia

MSC2020: 93-04
Research article
Full text in Russian

Received November 6, 2024
Revised November 15, 2024
Accepted November 30, 2024

This paper uses the model checking method for an exact schedulability test of real-time systems running on multiprocessor platforms. To use this method, we formally describe real-time systems with an abstract scheduler as Kripke models. This formalization provides terms sufficient to specialize the abstract scheduler. We illustrate our approach by explicitly defining schedulers that take into account preemption/non-preemption of tasks and global fixed or earliest-deadline-first priority in various combinations. The safety (schedulability) property of real-time systems is formulated using linear temporal logic LTL. Formalizing real-time systems as Kripke models and specifying the safety (schedulability) property as an LTL formula allows us to reduce the exact schedulability test of such systems to a model checking problem. We validate this approach to an exact schedulability test by implementing our formalization of real-time systems with non-preemptive global fixed-priority (NP-GFP), preemptive global fixed-priority (P-GFP), non-preemptive earliest-deadline-first priority (NP-EDF), and preemptive earliest-deadline-first priority (P-EDF) schedulers in Promela, the input language of the model checking tool SPIN. We conduct experiments in SPIN to prove/disprove the safety (schedulability) property to evaluate the effectiveness of our approach. We propose a heuristic assessment of the schedulability of a real-time system based on the provability of unsafety and unprovability of safety of a real-time system executed on multiprocessor platforms with the number of processors differing by one.

Keywords: real-time systems; exact schedulability test; Kripke models; model checking; Promela; SPIN

INFORMATION ABOUT THE AUTHORS

Garanina, Natalia O. (corresponding author) | ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su
PhD, Senior researcher

For citation: N. O. Garanina, "An Exact Schedulability Test for Real-Time Systems with Abstract Scheduler on Multiprocessor Platforms", *Modeling and Analysis of Information Systems*, vol. 31, no. 4, pp. 474–494, 2024. DOI: 10.18255/1818-1015-2024-4-474-494.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 4, 2024
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Точный тест планируемости для систем реального времени с абстрактным планировщиком на мультипроцессорных платформах

Н. О. Гаранина[1]

DOI: 10.18255/1818-1015-2024-4-474-494

[1]Институт автоматики и электрометрии СО РАН, Новосибирск, Россия

УДК 004.415.52
Научная статья
Полный текст на русском языке

Получена 6 ноября 2024 г.
После доработки 15 ноября 2024 г.
Принята к публикации 30 ноября 2024 г.

Эта статья посвящена использованию метода верификации моделей для точного теста планируемости систем реального времени, выполняющихся на мультипроцессорных платформах. Чтобы использовать этот метод, мы формально описываем системы реального времени с абстрактным планировщиком, используя модели Крипке. Эта формализация содержит термины, достаточные для специализации абстрактного планировщика. Мы иллюстрируем наш подход, явно определяя планировщики, которые учитывают вытеснение/невытеснение задач и глобальный фиксированный приоритет или приоритет ближайшего дедлайна в различных сочетаниях. Свойство безопасности (планируемости) систем реального времени сформулировано с помощью линейной темпоральной логики LTL. Формализация систем реального времени как моделей Крипке и задание свойства безопасности (планируемости) как формулы LTL позволяет свести точный тест планируемости таких систем к задаче верификации моделей. Мы апробируем этот подход к точному тесту планируемости, реализуя на Promela — входном языке инструмента верификации моделей SPIN — нашу формализацию систем реального времени с невытесняющим планировщиком с глобальным фиксированным приоритетом (NP-GFP), вытесняющим планировщиком с глобальным фиксированным приоритетом (P-GFP), невытесняющим планировщиком с приоритетом ближайшего дедлайна (NP-EDF) и вытесняющим планировщиком с приоритетом ближайшего дедлайна (P-EDF). Мы проводим эксперименты в SPIN для доказательства/опровержения свойства безопасности (планируемости), чтобы оценить эффективность нашего подхода. Мы предлагаем эвристическую оценку планируемости системы реального времени на основе доказуемости небезопасности и недоказуемости безопасности системы реального времени при выполнении на мультипроцессорных платформах с числом процессоров, отличающимся на единицу.

Ключевые слова: системы реального времени; точный тест на планируемость; модели Крипке; проверка моделей; Promela; SPIN

ИНФОРМАЦИЯ ОБ АВТОРАХ

Гаранина, Наталья Олеговна (автор для корреспонденции) | ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su
Канд. физ.-мат. наук, старший научный сотрудник

Для цитирования: N. O. Garanina, “An Exact Schedulability Test for Real-Time Systems with Abstract Scheduler on Multiprocessor Platforms”, *Modeling and Analysis of Information Systems*, vol. 31, no. 4, pp. 474–494, 2024. DOI: 10.18255/1818-1015-2024-4-474-494.

Введение

Классические системы реального времени, впервые достаточно полно описанные в [1], представляют собой множество задач, работы которых выполняются C единиц времени и должны завершиться в течение D единиц времени с момента запуска, причём запуск работ одной и той же задачи может происходить не чаще, чем один раз в P единиц времени. В современном мире эти системы встречаются буквально на каждом шагу — это и встроенные системы, и системы интернета вещей, и технологические процессы, и автоматические системы управления в автомобилестроении, авионике, космической отрасли и т. п. Эти задачи могут использовать ресурсы одного или нескольких процессоров. Как правило, процессоров значительно меньше, чем задач, поэтому естественным образом возникает вопрос планирования их выполнения. Существуют различные способы задания планировщиков в зависимости от предметной области. Например, в некоторых случаях можно допускать прерывание исполняющихся задач, а в других случаях такой подход может приводить к поломке системы. Основным вопросом для систем, описанных в терминах времени выполнения C, относительного дедлайна D и периодичности P, является вопрос о безопасности (планируемости): действительно ли в данной системе задач с фиксированными характеристиками и заданным планировщиком ни одна задача никогда не пропустит свой дедлайн?

Эту задачу решают много лет для различных систем и различных планировщиков. Для систем, в которых всего один процессор, задача исследована достаточно хорошо [1]. Однако для многопроцессорных систем возникает проблема очень большого количества вариантов поведения задач и точные методы проверки свойства безопасности оказываются слабо применимыми в явном виде. Были разработаны подходы, основанные на аппроксимации худшего случая исполнения [2], однако точные тесты на планируемость являются более предпочтительными для систем реального времени, критичных по безопасности и ресурсоемкости, и соответствующие подходы продолжают развиваться [3—8].

Наряду с разработкой специализированных методов точной проверки планируемости, есть ряд работ, использующих общие формальные методы анализа программ и систем [9]. Общие формальные методы могут предоставить модели, алгоритмы и возможности точного решения основной задачи безопасности систем реального времени, не представленные в специализированных подходах. Например, в методе верификации моделей [10] для снижения ресурсоемкости обхода графа состояний системы, предложены такие методы как редукция частичных порядков (используемая в инструменте SPIN [11]) или символьное представление состояний в виде упорядоченных бинарных решающих диаграмм ROBDD (применяющееся в инструменте nuXmv [12]). Настоящая работа развивает направление применения общих формальных методов для проверки безопасности систем реального времени в контексте метода верификации моделей. Рассмотрим некоторые работы по применению формальных методов в исследуемой области. В работах [13, 14] исследовались представление планировщика с фиксированным глобальным приоритетом для многопроцессорной системы и систем невытесняемых самоприостанавливающихся задач, с помощью временных автоматов. В [15] моделировался частный случай системы на языке Promela верификатора SPIN [11], а в работе [16] представлена Promela-модель для однопроцессорной системы. В работе [17] использовались игры на графах для упрощения задачи достижимости в точном тесте планируемости.

В нашей работе мы формально представляем системы реального времени с абстрактным планировщиком как модели Крипке, которые используются для верификации параллельных и распределенных систем методом проверки моделей. Такая формализация позволяет использовать возможности этого метода для проверки безопасности систем реального времени. При этом специализация абстрактного планировщик в терминах предложенной модели Крипке дает возможность получать конкретные системы реального времени. Возможность задавать различные виды планировщиков в рамках одной модели отличает нашу работу от перечисленных выше, где и систем,

и планировщики зафиксированы. Мы приводим пример специализации абстрактного планировщика для невытесняющего планировщика с глобальным фиксированным приоритетом, вытесняющего планировщика с глобальным фиксированным приоритетом, невытесняющего планировщика с приоритетом ближайшего дедлайна и вытесняющего планировщика с приоритетом ближайшего дедлайна. Кроме того, мы формулируем свойство безопасности системы реального времени в терминах линейной темпоральной логики LTL [10]. Нашу формализацию систем реального времени со всеми четырьмя планировщиками мы реализуем на языке Promela верификатора SPIN и проводим ряд экспериментов по доказательству или опровержению свойства безопасности для выяснения производительности нашего метода.

Оставшаяся часть статьи имеет следующую структуру. В разделе 1 мы напоминаем базовые определения систем реального времени и планирования, формализуем системы реального времени как модели Крипке, а также формально определяем четыре различных планировщика. Раздел 2 рассматривает особенности языка Promela и описывает модель Promela для системы реального времени с определенными выше четырьмя различными планировщиками. В разделе 3 описаны эксперименты с предложенной Promela-моделью системы реального времени. В заключении мы подводим итоги и представляем планы развития настоящей работы.

1. Модель Крипке реального времени

Мы считаем, что *система реального времени* является набором задач $T = (T_1, \ldots, T_n)$, где каждая *задача* $T_i = (C_i, D_i, P_i)$ имеет *время выполнения* C_i, *относительный дедлайн* D_i и *минимальный период* P_i. Каждая задача $T_i \in T$ может запускать потенциально бесконечное количество работ, каждая из которых требует C_i единиц времени для выполнения. Эти работы должны быть завершены до D_i единиц времени после времени запуска. Моменты времени запуска разделены как минимум P_i единицами времени. Если нет других ограничений на запуски заданий, эти задачи называются *спорадическими*[1] *задачами*. В этой статье мы изучаем базовый случай систем реального времени, в которых все параметры задач являются натуральными числами больше нуля и при этом $C_i < D_i \leq P_i$. Все работы задач выполняются на m процессорах. Если число процессоров меньше числа задач ($m < n$), нам нужен *планировщик*, чтобы решить, работе какой задачи можно предоставить свободный процессор для выполнения. *Задача проверки планируемости* заключается в том, чтобы определить, завершаются ли все работы каждой задачи в системе реального времени до дедлайна. Мы также будем называть систему реального времени *безопасной*, если проверка планируемости даёт положительный результат. Если такая проверка даёт отрицательный результат, система *небезопасна*. Для оставшейся части статьи мы фиксируем указанную выше систему реального времени T, n задач которой выполняются на m процессорах.

Планировщик должен точно определить условия допуска работ для их выполнения на свободных процессорах. Существует много типов планировщиков, использующих разные условия допуска и их сочетания. Некоторые условия перечислены ниже.

- Глобальный фиксированный приоритет (GFP, Global Fixed Priority). Набор задач упорядочен: T_1 имеет наивысший приоритет, T_n имеет низший приоритет, и работа задачи с меньшим номером имеет приоритет допуска на выполнение над работой задачи с большим номером.
- Приоритет ближайшего дедлайна (EDF, Early Deadline-First). Работа с более ранним дедлайном имеет более высокий приоритет допуска на выполнение.
- Приоритет наименьшего резерва (LST, Least-Slack-Time-First). Работа с наименьшим резервом времени на выполнение до дедлайна, имеет более высокий приоритет допуска на выполнение.

[1] В фундаментальной книге [1] такие задачи называются периодическими, однако здесь мы будем использовать более распространенную сейчас терминологию.

- Невытеснение. Ни одна работа не может быть прервана другой (даже работой более приоритетной задачи).
- Вытеснение. Работа может быть прервана другой работой более приоритетной задачи.
- Динамическое планирование. В многопроцессорной системе работы каждой задачи могут выполняться на разных процессорах (включая часть работы, оставшуюся после вытеснения).
- Статическое планирование. В многопроцессорной системе все работы каждой задачи выполняются на одном заранее определенном процессоре.
- Онлайн планирование. Планировщик принимает решение о допуске работы на выполнение в момент её запуска или в момент завершения другой работы.
- Офлайн планирование. Планировщик принимает решение о допуске работ на выполнение до начала работы системы.

В нашей работе мы сосредоточимся на динамических онлайн планировщиках и будем называть их *абстрактными планировщиками* в смысле абстрагирования от конкретного способа присваивания приоритетов и вытесняемости. Далее мы формально специализируем абстрактный планировщик, рассматривая как глобальный фиксированный приоритет, так и приоритет ближайшего дедлайна в сочетании с вытесняемыми и невытесняемыми задачами.

Модели Крипке используются, в частности, в формальных методах верификации моделей программ и систем [10]. В настоящей статье мы будем их применять в этом контексте, сводя задачу проверки планируемости к задаче верификации модели Крипке реального времени относительно свойства планируемости, выраженного с помощью темпоральной логики линейного времени LTL [18]. Напомним определение модели Крипке. Пусть задано множество элементарных высказываний *Prop*. *Модель Крипке* — это набор $M = (S, S_0, R, L)$, где

- S — конечное множество состояний S;
- $S_0 \subseteq S$ — множество начальных состояний;
- $R \subseteq S \times S$ — тотальное отношение переходов между состояниями;
- $L : Prop \longrightarrow 2^S$ — функция означивания, связывающая состояния модели и истинность элементарных высказываний.

В нашей модели Крипке реального времени для моделирования абстрактного планировщика мы используем предикат $go(i, s)$, который равен *true*, если задача i может начать выполнять работу в состоянии s, и *false* в противном случае. Далее мы уточняем $go(i, s)$ для невытесняющего планировщика с глобальным фиксированным приоритетом (NP-GFP), вытесняющего планировщика с глобальным фиксированным приоритетом (P-GFP), невытесняющего планировщика с приоритетом ближайшего дедлайна (NP-EDF) и вытесняющего планировщика с приоритетом ближайшего дедлайна (P-EDF).

По мотивам статьи [19], введём текущие значения параметров задач следующим образом для задания состояний модели Крипке. Для каждой задачи $i \in T$ набор $s_i = (i, C'_i, D'_i, P'_i, rel_i, go_i, end_i, bad_i)$ — это состояние задачи i, где

- $C'_i \leq C_i$ — время, оставшееся до завершения работы этой задачи;
- $D'_i \leq D_i$ — время до дедлайна выполнения работы этой задачи;
- $P'_i \leq P_i$ — время до следующего допустимого запуска работы этой задачи;
- $rel_i \in \mathbb{B}$ — булева переменная, которая отмечает запуск и последующую необходимость выполнения работы: она становится *true*, когда работа задачи i запускается, и становится *false*, когда эта работа завершена.
- $go_i \in \mathbb{B}$ — булева переменная, которая отмечает допуск работы к выполнению: она становится *true*, когда работа задачи i допущена планировщиком к выполнению, и становится *false*, когда эта работа завершена или этот допуск отозван при вытеснении;

- $end_i \in \mathbb{B}$ — булева переменная, которая отмечает завершение работы: она становится *true*, когда работа задачи i только что завершилась, и ее значение *false* все остальное время;
- $bad_i \in \mathbb{B}$ — булева переменная, которая отмечает, что работа пропускает свой дедлайн: она *false*, если $C'_i \le D'_i$, и становится *true* в противном случае.

Булевы переменные rel_i, go_i и end_i необходимы для моделирования динамических онлайн планировщиков, поскольку решения о планировании принимаются в дискретные моменты времени, начиная с 0 при запуске или завершении работ.

Назовём *изменением статуса работы* её запуск или завершение. Поскольку работа планировщика происходит в момент изменения статуса работы, в модели Крипке между переходами, моделирующими течение времени (изменения текущих значений параметров задач) должны быть переходы для изменения статуса работ, чтобы в следующем переходе по времени работам была известна реакция планировщика на это событие. Для моделирования такого чередования введём в представление в системе задачи i булеву переменную st_i, которая принимает значение *true*, если работа только что запущена или если работа только что завершилась, в остальных случаях значение этой переменной *false*, что соответствует выполнению работы или пребыванию задачи i в спящем режиме. Пусть $s.st_i$ — проекция состояния s системы на переменную st_i задачи i. Введём на состояниях системы предикат $Plan(s) = \bigvee_{i=1}^{n} s.st_i$, который будет характеризовать глобальные переходы: если $Plan(s) = true$, то это означает, что есть работа, которая изменила свой статус и сейчас требуется решение планировщика о допуске её к выполнению или о вытеснении, а $Plan(s) = false$ в случае, если решение планировщика в данный момент времени не требуется.

Пусть степень загрузки процессоров в системе представлена переменной *busy*: это количество работ, которые в данный момент выполняются ($busy \le m$). Для расчёта изменения загрузки процессора *busy* нам также необходимо вычислить число изменения lc — количество задач, работы которых только что завершены или работы которых только что допущены к выполнению планировщиком. Для подсчёта этого числа мы будем интерпретировать булевы значения go_i и end_i как целые числа (1 для *true* и 0 для *false*), и тогда $lc = \sum_{i=1}^{n}(go_i - end_i)$.

Для понятности изложения, в состояниях системы мы заменим истинностные значения булевых переменных их именами с отрицанием или без него: *name* при $name = true$ и $\neg name$ при $name = false$ для $name \in \{st_i, rel_i, go_i, end_i, bad_i\}$. Когда переменная имеет произвольное значение истинности, будем писать $\mathbb{B}$. Пусть *Prop* — набор атомарных высказываний, состоящих из булевых комбинаций арифметических сравнений текущих значений параметров задач, утверждений о количестве запущенных работ и булевых статусах задач. Определим систему реального времени T с абстрактным планировщиком как *модель Крипке реального времени* $M^T = (S^T, s_0^T, R^T, L^T)$, где

- конечное множество состояний $S^T = \prod_{i=1}^{n}(\{i\}\times[0..C_i]\times[0..D_i]\times[0..P_i]\times\mathbb{B}\times\mathbb{B}\times\mathbb{B}\times\mathbb{B}\times\mathbb{B})\times[0..m]$; для глобального состояния $s \in S^T$, $s_i = (i, C'_i, D'_i, P'_i, st_i, rel_i, go_i, end_i, bad_i)$ — *проекция s на задачу i*, и $s.C'_i$, $s.D'_i$, $s.P'_i$, $s.rel_i$, $s.go_i$, $s.end_i$, $s.bad_i$ и $s.busy$ — проекции s на свои компоненты;
- начальное состояние $s_0^T = \prod_{i=1}^{n}\{(i, C_i, D_i, P_i, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)\} \times \{0\}$;
- тотальное отношение переходов $R^T \in S^T \times S^T$ определяется композицией i-проекций глобальных состояний s и t:
  $(s, t) \in R^T$, если и только если $t.busy = s.busy + \sum_{i=1}^{n}(go_i - end_i)$ и выполняется одно из следующих условий:
  1) $s_i = (i, C_i, D_i, P_i, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$, и $Plan(s) = false$, тогда
     (a) $t_i = (i, C_i, D_i, P_i, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — задача i ничего не делает или
     (b) $t_i = (i, C_i, D_i, P_i, st_i, rel_i, \neg go_i, \neg end_i, \neg bad_i$ — задача i запустила работу;
  2) $s_i = (i, C'_i, D'_i, P'_i, st_i, rel_i, \neg go_i, \neg end_i, \neg bad_i)$, при $0 < C'_i \le D'_i$ и $Plan(s) = true$
     (a) если $go(i, s)$, то $t_i = (i, C'_i - 1, D'_i - 1, P'_i - 1, \neg st_i, rel_i, go_i, \neg end_i, \neg bad_i)$ — работа задачи i запущена, и она начала (возобновила) выполнение;

(b) если $\neg go(i,s)$, то $t_i = (i, C'_i, D'_i - 1, P'_i - 1, \neg st_i, rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — работа задачи i запущена, но не выполняется;

3) $s_i = (i, C'_i, D'_i, P'_i, \neg st_i, rel_i, \mathbb{B}, \neg end_i, \neg bad_i)$, при $0 < C'_i \leq D'_i$ и
   (a) если $Plan(s) = false$ и,
      (i) если $s.go_i$, то $t_i = (i, C'_i - 1, D'_i - 1, P'_i - 1, \neg st_i, rel_i, go_i, \neg end_i, \neg bad_i)$ — работа задачи i продолжает выполнение;
      (ii) если $\neg s.go_i$, то $t_i = (i, C'_i, D'_i - 1, P'_i - 1, \neg st_i, rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — работа задачи i запущена, но не выполняется;
   (b) если $Plan(s) = true$ и
      (i) если $go(i,s)$, то $t_i = (i, C'_i, D'_i, P'_i, \neg st_i, rel_i, go_i, \neg end_i, \neg bad_i)$ — работа задачи i продолжает выполнение;
      (ii) если $\neg go(i,s)$, то $t_i = (i, C'_i, D'_i, P'_i, \neg st_i, rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — работа задачи i приостанавливает выполнение;

4) $s_i = (i, 0, D'_i, P'_i, \neg st_i, rel_i, go_i, \neg end_i, \neg bad_i)$, при $0 \leq D'_i$ и $Plan(s) = false$,
   (a) если $D'_i > 0$, то $t_i = (i, C_i, D'_i, P'_i, st_i, \neg rel_i, \neg go_i, end_i, \neg bad_i)$ — работа задачи i завершена, но время до дедлайна не исчерпано;
   (b) если $D'_i = 0$ и $P'_i > 0$, то $t_i = (i, C_i, D_i, P'_i, st_i, \neg rel_i, \neg go_i, end_i, \neg bad_i)$ — работа задачи i завершена и время до дедлайна исчерпано, но время для следующего возможного запуска ещё не наступило;
   (c) если $P'_i = 0$, то $t_i = (i, C_i, D_i, P_i, st_i, \neg rel_i, \neg go_i, end_i, \neg bad_i)$ — работа задачи i завершена, время до дедлайна исчерпано и наступило время для следующего возможного запуска;

5) $s_i = (i, C_i, D'_i, P'_i, st_i, \neg rel_i, \neg go_i, end_i, \neg bad_i)$ и $Plan(s) = true$, то
   $t_i = (i, C_i, D'_i, P'_i, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — планировщик фиксирует завершение работы задачи i;

6) $s_i = (i, C_i, D'_i, P'_i, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$, при $0 \leq D'_i < D_i$ и $Plan(s) = false$,
   (a) если $D'_i > 0$, то $t_i = (i, C_i, D'_i - 1, P'_i - 1, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — работа задачи i завершена, но время до дедлайна не исчерпано;
   (b) если $D'_i = 0$ и $P'_i > 0$, то $t_i = (i, C_i, D_i, P'_i - 1, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — работа задачи i завершена и время до дедлайна исчерпано, но время для следующего возможного запуска ещё не наступило;
   (c) если $P'_i = 0$, то $t_i = (i, C_i, D_i, P_i, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — работа задачи i завершена, время до дедлайна исчерпано и наступило время следующего возможного запуска;

7) $s_i = (i, C_i, D_i, P'_i, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$, при $0 \leq P'_i < P_i$ и $Plan(s) = false$,
   (a) если $P'_i > 0$, то $t_i = (i, C_i, D_i, P'_i - 1, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — работа задачи i завершена и время до дедлайна исчерпано, но время для следующего возможного запуска ещё не наступило;
   (b) если $P'_i = 0$, то $t_i = (i, C_i, D_i, P_i, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, \neg bad_i)$ — работа задачи i завершена, время до дедлайна исчерпано и наступило время следующего возможного запуска;

8) $s_i = (i, C'_i, C'_i - 1, P'_i, \neg st_i, rel_i, \mathbb{B}, \neg end_i, \neg bad_i)$, $Plan(s) = false$, то
   $t_i = (i, 0, 0, 0, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, bad_i)$ — работа задачи i пропускает свой дедлайн, так как ей не хватает времени для выполнения оставшейся части работы;

9) $s_i = (i, 0, 0, 0, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, bad_i)$ и $t_i = (i, 0, 0, 0, \neg st_i, \neg rel_i, \neg go_i, \neg end_i, bad_i)$ — задача i бесконечно пребывает в плохом состоянии;

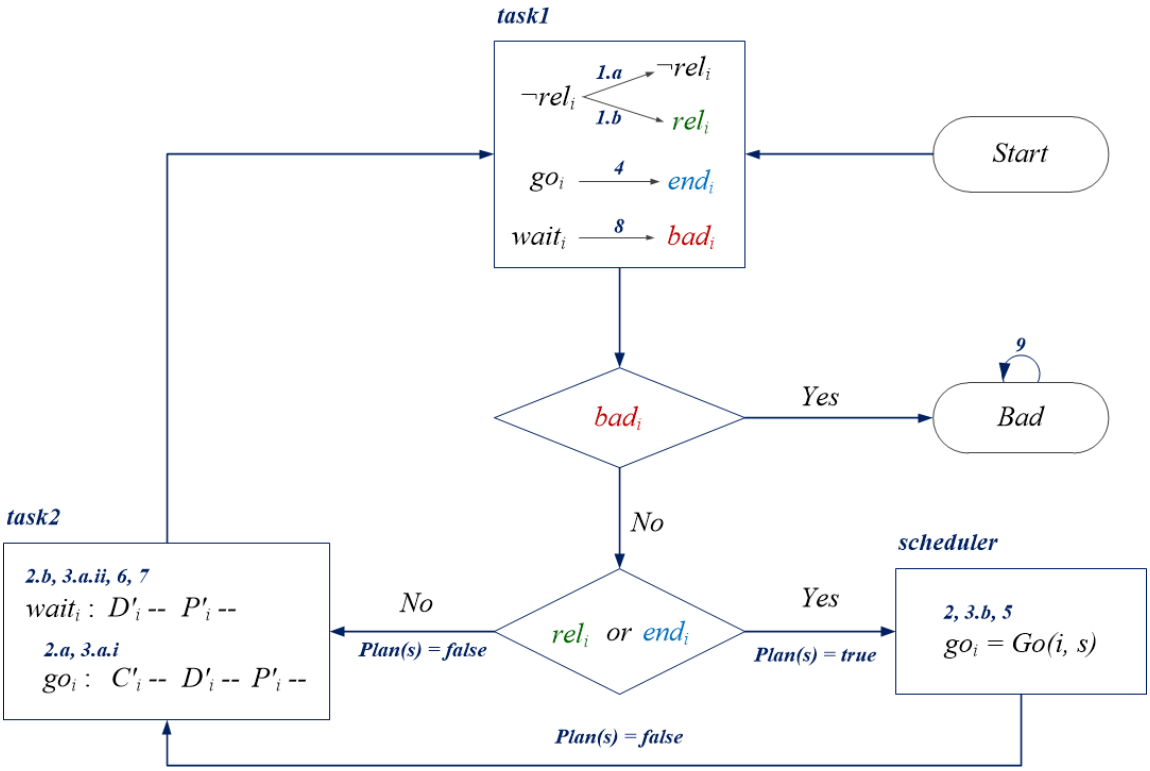

Fig. 1. Graphical representation of a real-time Kripke model with an abstract scheduler

Рис. 1. Графическое представление модели Крипке реального времени с абстрактным планировщиком

- функция означивания $L : Prop \longrightarrow 2^{S^T}$ стандартна: она сопоставляет выражениям сравнения переменных модели Крипке реального времени и статусов задач те состояния, в которых они истинны.

Вышеописанная модель представлена в упрощенном графическом виде на рис. 1. После запуска в *Start*, безопасная модель Крипке реального времени функционирует в бесконечном цикле, не попадая в плохое состояние *Bad*. В этом цикле в блоке *task1* может произойти изменение статуса задачи: она может запустить работу (пункт 1.*b* в списке условий R^T), завершить работу (пункт 4 в списке условий R^T) или обнаружить, что она пропускает дедлайн (пункт 8 в списке условий R^T). После этого проверяется, есть ли пропущенные дедлайны. Если они есть, то соответствующая задача (и, можно сказать, что система в целом) переходит в состояние *Bad*. В противном случае, проверяется наличие только что запущенных или только что завершившихся работ. Если они есть, то выполняется блок *scheduler*, в котором вычисляется возможность выполнения работы, запущенной только что или ранее в соответствии с пунктами 2, 3.*b* и 5 в списке условий R^T. После этого, а также в случае отсутствия только что запущенных или завершившихся работ, выполняется блок *task2*. В этом блоке убывают текущие значения относительного дедлайна и периода ожидающих работ (пункты 2.*b*, 3.*a.ii*, 6 и 7 в списке условий R^T) и убывают эти же значения и время оставшееся до завершения выполняющихся работ (пункты 2.*a*, и 3.*a.i* в списке условий R^T). Отметим, что после достижения нулевого значения текущим относительным дедлайном до истечения периода, он перестает убывать, но чтобы не загромождать схему, мы опустили эти подробности. После изменения значений текущих параметров работ, выполнение снова переходит в блок *task1*.

Для специализации абстрактного планировщика нам необходимо определить предикат $go(i,s)$. Этот предикат для всех типов планировщиков использует информацию о загрузке процессоров *busy*, задачах и работах системы: их параметрах, приоритетах, времени с начала запуска, времени до дедлайнов и т. д. Вся эта информация доступна в состояниях системы, поэтому $go(i,s)$ можно сформулировать в терминах вышеопределнной модели Крипке реального времени. Рассмотрим специализацию четырех вариантов планировщиков: невытесняющего планировщика с глобальным фиксированным приоритетом (NP-GFP), вытесняющего планировщика с глобальным фиксированным приоритетом (P-GFP), невытесняющего планировщика с приоритетом ближайшего дедлайна (NP-EDF) и вытесняющего планировщика с приоритетом ближайшего дедлайна (P-EDF).

1. *Невытесняющий планировщик с глобальным фиксированным приоритетом (NP-GFP).*
Предикат $go(i,s) \equiv (|MajGFP(i,s)| + s.busy < m) \vee s.C'_i < C_i$, где $MajGFP(i,s) = \{j \in [1..n] \mid j < i \wedge s.rel_j \wedge \neg s.go_j\}$ — это множество запущенных заданий с более высоким GFP-приоритетом, которые ещё не начали выполняться. Отметим, что при $s.C'_i < C_i$ работа задачи i выполняется и поскольку в этом случае $go(i,s) = true$, она продолжает выполняться, т. е. не вытесняется.

2. *Вытесняющий планировщик с глобальным фиксированным приоритетом (NP-GFP).*
Предикат $go(i,s) \equiv |MajGFP(i,s)| < m$. В этом случае, если количество работ задач с более высоким приоритетом обеспечивает полную загрузку всех процессоров, то работа задачи i не будет выполняться, даже если она уже исполнялась на каком-либо процессоре, так как условие текущего выполнения не учитывается в предикате $go(i,s)$.

3. *Невытесняющий планировщик с приоритетом ближайшего дедлайна (NP-EDF).*
Предикат $go(i,s) \equiv (|MajEDF(i,s)| + s.busy < m) \vee s.C'_i < C_i$, где $MajEDF(i,s) = \{j \in [1..n] \mid D'_j < D'_i \wedge s.rel_j \wedge \neg s.go_j\}$ — это множество запущенных заданий с более высоким EDF-приоритетом, которые еще не начали выполняться. Здесь, как и для NP-GFP планировщика, учитывается условие $s.C'_i < C_i$, обеспечивающее невытеснение выполняющейся работы задачи i.

4. *Вытесняющий планировщик с приоритетом ближайшего дедлайна (P-EDF).*
Предикат $go(i,s) \equiv |MajEDF(i,s)| < m$. Аналогично P-GFP планировщику, работа задачи i вытесняется (или не запускается), если все процессоры заняты работами более приоритетных задач, а разница здесь лишь в определении мажорирующего множества.

Таким образом, можно определить абстрактные относительно приоритетов предикаты

- для невытесняющего планировщика $go_{NP}(i,s) \equiv (|Maj(i,s)| + s.busy < m) \vee s.C'_i < C_i$ и
- для вытесняющего планировщика $go_P(i,s) \equiv |Maj(i,s)| < m$,

где $Maj(i,s)$ — множество работ более приоритетных задач в соответствии с выбранным способом установления приоритетов.

Пометим состояние системы реального времени T с $D'_i = C'_i - 1$ для задачи i как *плохое состояние*, потому что при таком условии у этой задачи не осталось времени, чтобы завершить работу до дедлайна. *Плохие состояния в модели Крипке реального времени* M^T — это множество $Bad\_States = \{s \in S^T \mid \exists i \in [1..n] : \ s.bad_i = \mathbf{1}\}$. Высказывание $bad = \bigvee_{i=0}^{n}(s.bad_i = \mathbf{1})$ описывает это множество. Следовательно, точный тест на планируемость для системы реального времени T заключается в проверке того, выполнима ли в модели Крипке реального времени M^T формула линейной темпоральной логики LTL $\Phi_T = \mathbf{G}(\neg bad)$, которая читается как «Always not *bad*», то есть *система реального времени никогда не достигает состояния, в котором какая-либо задача находится в плохом состоянии.*

2. Модель Promela для систем реального времени с вытесняющими и невытесняющими GFP- и EDF-планировщиками.

В этом разделе мы описываем реализацию модели реального времени M^T для системы реального времени T в Promela — входном языке проверки моделей SPIN. Язык Promela используется для описания параллельных взаимодействующих процессов на основе формализма CSP [20]. Программа

Promela состоит из параллельных процессов, общающихся через каналы или общие переменные. Выполнение множества параллельных процессов Promela использует семантику чередования. Чередование может быть ограничено операторами `atomic` и `d_step`, которые не разрешают прерывание указанной последовательности действий процесса. Язык Promela включает блокирующие операторы управления `if` и `do`. Кроме того, для улучшения читаемости и сокращения кода в языке Promela можно использовать блоки `inline`, которые при интерпретации помещают последовательность операторов внутри этого блока в место «вызова» блока, и фактически могут работать как функция (нерекурсивная), поскольку содержат параметрические переменные, заменяемые при интерпретации фактическими переменными в месте «вызова». Модель Promela может быть проверена с помощью верификатора моделей SPIN на соответствие требованиям, выраженных в терминах линейной темпоральной логики LTL, поэтому она предполагает только конечные типы для переменных модели. Модель реального времени M^T имеет конечное число состояний, и ее представление в Promela не требует абстракции данных.

Мы моделируем M^T в Promela для выполнения точного теста на планируемость, то есть для определения того, завершается ли каждая работа каждой задачи до своего дедлайна. Promela-модель реализует абстрактный планировщик как один из четырёх видов планировщиков, для которых мы описали специализацию предиката $go(i, s)$. Для простоты мы считаем, что период P_i равен D_i для каждого i[2]. Ниже мы приводим подробности реализации, опуская часть кода для краткости (в частности, мы не приводим объявления переменных модели).

Инициализирующий Promela-процесс активирует процесс `tasks`, моделирующий поведение множества задач и один из процессов-планировщиков NP-GFP, P-GFP, NP-EDF или P-EDF. Процессы-планировщики, не входящие в проверяемую систему, закомментированы.

Листинг 1: Promela-процесс для запуска системы

```
1  init{
2  atomic{
3       run tasks();
4       run schedulerNPGFP();
5  //   run schedulerPGFP();
6  //   run schedulerNPEDF();
7  //   run schedulerPEDF();
8    }
9  }
```

2.1. Моделирование поведения задач

Процесс `tasks`, приведенный в листинге 2, сначала задает для `NumTask` задач начальные значения их текущих параметров C'_i и D'_i, присваивая элементам глобальных массивов `C_cur` и `D_cur` синтетические значения в строках 4 и 5. Конкретные не синтетические системы реального времени можно моделировать явным присваиваниям нужных значений элементам этих массивов и в `inline`-вставках строк 11 и 22. В строках процесса 6 и 8 подсчитывается значение утилизации $U = (\sum_{i=1}^{n} \frac{C_i}{D_i})/m$, которая отражает максимальную нагрузку множества m процессоров системы для заданного множества n задач. Это значение нужно для оценки производительности планировщиков: чем выше значение утилизации системы, которая оказывается безопасной при управлении данным планировщиком, тем этот планировщик лучше. Поскольку Promela допускает только целочисленное деление, мы подсчитываем значение утилизации в переменной `Util` сразу в процентах, умножая отношение времени работы и периода выполнения на 100. В строке 7 инициализируется очередь запущенных, но не выполняющихся процессов, которая используется процессами динамических планировщиков NP-EDF и P-EDF, описанными ниже.

[2]Такое упрощение является распространенной практикой при задании и анализе систем реального времени, так как часто не требуется, чтобы работа была завершена раньше, чем она сможет начаться снова.

В предыдущей версии Promela-модели для систем реального времени, описанной в [21], задачи системы задавались как отдельные процессы, что несколько повышало ресурсоемкость верификации за счёт их интерливинга, а также усложняло само описание системы за счёт необходимости строгой синхронизации всех процессов, как на уровне процессов для задач, так и на уровне планировщика. На самом деле в данной системе все процессы задач действуют синхронно, то есть одновременно делают свой шаг вычисления, запуская и завершая работу, либо уменьшая значение времени до дедлайна D'_i в `D_cur[i]`, а, возможно, и значение времени до конца выполнения C'_i в `C_cur[i]`. Поэтому нет необходимости в недетерминированном чередовании исполнения процессов задач и текущая версия Promela-модели для систем реального времени реализует выполнения шагов задач посредством последовательных `inline`-вставок шагов задач в строках 11 и 22.

После вычисления значения утилизации система начинает свою работу в цикле (строки 9–24), который является бесконечным, если система безопасна, то есть если значение переменной `BAD`, фиксирующей попадание некоторой задачи в плохое состояние, всегда ложно. Последовательность действий процесса `tasks` соответствует модели Крипке реального времени M^T в части определения необходимости принятия решений о планировании и отражает «крупные» шаги поведения системы реального времени с планировщиком:

1. В строке 11 последовательность `inline`-вставок `task_plan` с синтетическими значениями параметров задач моделирует запуск, завершение работ системы или пропуск дедлайна, получая информацию о необходимости принятия решений планировщиком в булевой переменной `plan`, соответствующей предикату $Plan(s)$ модели M^T.
2. Если принятие решения планировщиком необходимо (строка 16), то процесс задач отправляет запрос процессу-планировщику через рандеву-канал `task_shed` (строка 17) и после его отклика в строке 19 переустанавливает переменную `plan`.
3. В строке 22 с помощью последовательности `inline`-вставок `task_step` с синтетическими значениями параметров задач моделируется само выполнение задач системы реального времени.

Листинг 2: Promela-процесс для задач

```
1  proctype tasks() {
2  atomic{
3    for(i : 0 .. NumTask-1){
4        C_cur[i] = i+1;
5        D_cur[i] = 2*(i+2);
6        Util = Util + (100*(i+1))/(2*(i+2));
7        que[i] = 255; }
8    Util = Util/NumProc; }
9  do
10   :: atomic{
11         for (i : 0 .. NumTask-1){ task_plan(i, i+1, 2*(i+2)); }
12         if
13         :: BAD -> break;
14         :: else ->
15                 if
16                 :: plan ->
17                      task_shed ! true;
18                      task_shed ? false;
19                      plan = false;
20                 :: else -> skip;
21                 fi
22                 for (i : 0 .. NumTask-1){ task_step(i, i+1, 2*(i+2)); }
23         fi }
24 od }
```

Рассмотрим, как устроена `inline`-вставка `task_plan(me,C,D)`, которая моделирует запуск, завершение отдельной работы системы, относящейся к задаче с номером `me`, временем выполнения `C` и относительным дедлайном `D` или пропуск ею дедлайна. Её Promela-код представлен на ли-

стинге 3. Следуя определению M^T, этот Promela-код реализуют отношение перехода R^T почти напрямую с учётом равенства относительного дедлайна и периода задач. Отметим, что для реализации «глобальной» информации, содержащейся в дизъюнкции переменных статуса задач st_i и bad_i, достаточно использовать изменения значений глобальных переменных `plan` и `BAD` одной из задач.

Строки 4–9 соответствуют пункту 1 определения отношения переходов R^T. При этом в случае, если задача запускает работу в строке 6, то она устанавливает значение переменной `plan` в *true*, что означает, что планировщик должен принять решение о выполнении этой задачи, если же задача не запускает работу, то значение этой переменной остаётся неизменным. Строки 10–25 соответствуют пункту 4 определения отношения переходов R^T, а если период задачи подошёл к концу, то и пункту 1. В любом случае, задача уменьшает значение нагрузки процессоров в строке 11, переустанавливает значения текущего остатка времени выполнения (строка 12), а, возможно, и остатка времени до дедлайна (строка 16). В массиве `go` содержится информация о выполняемых работах и задача изменяет эту информацию в строке 13. Так же в строке 23 задача фиксирует завершение своей работы в массиве `end`, значения элементов которого используют планировщики NP-EDF и P-EDF. И снова значение переменной `plan` устанавливается в *true*, так как произошло завершение работы. Если задача пропускает свой дедлайн, она устанавливает специальную булеву переменную `BAD` в *true* (строка 27).

Листинг 3: Inline-код для изменения статуса работы

```
1  inline task_plan(me, C, D){
2  atomic {
3     if
4     ::  C_cur[me] == C && D_cur[me] == D && !release[me] -> // release
5          if
6          :: release[me] = true; // point 1.b
7             plan = true;
8          :: skip;                // point 1.a
9          fi
10     :: C_cur[me] == 0 ->  // finish
11         busy--;
12         C_cur[me] = C;
13         go[me] = false;
14         if
15         :: D_cur[me] == 0 -> // point 4.b and 4.c
16             D_cur[me] = D;
17             if
18             :: release[me] = false; // point 1.a
19             :: skip;                // point 1.b
20             fi
21         :: else ->
22             release[me] = false;    // point 4.a
23             end[me] = true;
24         fi
25         plan = true;
26     :: C_cur[me] > D_cur[me] && release[me] -> // fail deadline
27         BAD = true; // point 8, 9
28         tmp = me;
29     :: else -> skip;
30     fi
31  }}
```

Код `inline`-вставки `task_step(me,C,D)`, представленной на листинге 4 моделирует «вычислительные» шаги отдельной работы системы, относящейся к задаче с номером `me`, временем выполнения `C` и относительным дедлайном `D`. Эти шаги состоят в уменьшении текущих значений остатка времени выполнения `C_cur` и остатка времени до дедлайна `D_cur` при условии, что работа не завершена и разрешение на выполнение `go[me]` имеет место (строка 7), а если разрешения нет,

то уменьшается только D_cur (строка 8). Если работа не выполняется, а время её очередного запуска ещё не пришло, то также изменяется D_cur (строки 10–17).

Листинг 4: Inline-код для шага работы

```
1  inline task_step(me, C, D){
2  atomic {
3     if
4     :: C_cur[me]>0 && D_cur[me]>0 && C_cur[me]<=D_cur[me] && release[me]
      -> // executing
5         if
6         ::  go[me] ->
7               C_cur[me]--; D_cur[me]--;  // point 3.a.i, executing job
8         :: else -> D_cur[me]--; // point 3.a.ii, waiting
9         fi
10    :: C_cur[me] == C && D_cur[me] > 0 && D_cur[me] < D && !release[me]
      ->
11        // points 6 and 7, time till the next release
12        D_cur[me]--;
13        if
14        :: D_cur[me] == 0 ->
15            D_cur[me] = D;
16        :: else -> skip;
17        fi
18     :: else -> skip;
19     fi
20  }
21 }
```

2.2. Моделирование поведения планировщиков

В нашей модели все планировщики принимают решение о возможности выполнения работ, в моменты времени, когда запускается или завершается очередная работа какой-либо задачи. Действия планировщиков помещены в блок atomic, последовательность действий которого SPIN рассматривает как один вычислительный шаг. Такое размещение снижает вычислительную сложность проверки модели.

Рассмотрим Promela-модель schedulerNPGFP() невытесняющего планировщика с глобальным фиксированным приоритетом (NP-GFP), представленную на листинге 5. В бесконечном (в случае безопасности системы) цикле планировщик NP-GFP при получении запроса на планирование в строке 3, подсчитывает количество свободных процессоров в переменной free в строке 5 и, в случае наличия свободных процессоров, в строке 11, проверив, что очередная по убыванию приоритета задача запустила работу, которая пока не выполняется, в строке 12 предоставляет ей возможность занять процессор, изменяя значение соответствующего элемента в массиве go. Затем планировщик увеличивает значение переменной загрузки процессоров busy, а количество свободных процессоров, напротив — уменьшает (строки 13–14). Когда свободные процессоры заканчиваются на некотором шаге i, планировщик в строке 8 завершает выдачу разрешений на выполнение, и поэтому запрашивающие разрешение процессы с более низким приоритетом, чем $i-1$, остаются в состоянии ожидания, так как значения соответствующих им элементов в массиве go остаются *false*. Планировщик в строке 18 возвращает управление процессу задач tasks.

Листинг 5: Promela-процесс для планировщика NP-GFP

```
1  proctype schedulerNPGFP(){
2   do
3   :: task_shed ? true ->
4       atomic{
5         free = NumProc - busy;
6         for (i : 0 .. NumTask - 1){
7              if
```

```
 8              :: free == 0 -> break;
 9              :: else ->
10                   if
11                   :: release[i] && !go[i] && free != 0 ->
12                        go[i] = true; // go!
13                        busy++;
14                        free--;
15                   :: else -> skip;
16                   fi
17              fi }
18         task_shed ! false; }
19   :: BAD -> break;
20   od;
21 }
```

Представленная на листинге 6 Promela-модель вытесняющего планировщика с глобальным фиксированным приоритетом (P-GFP) реализует вытеснение согласно заданным приоритетам вначале выдавая разрешение всем запросившим выполнение задачам в строках 7–9, а затем отзывая лишние разрешения в строках 16–17, что вызывает приостановку или не-запуск выполнения работы. Разрешение оказывается лишним, если значение переменной загрузки процессоров busy, возрастающей при обнаружении в строке 22 выданного разрешения, станет максимально возможным (строка 14).

Листинг 6: Promela-процесс для планировщика P-GFP

```
 1 proctype schedulerPGFP(){
 2  do
 3  :: task_shed ? true ->
 4       atomic{
 5         for (i : 0 .. NumTask - 1){
 6              if
 7              :: release[i] && !go[i] ->
 8                   go[i] = true; // go!
 9              :: else -> skip;
10              fi }
11         busy = 0;
12         for (i : 0 .. NumTask - 1){
13              if
14              :: busy == NumProc ->
15                   if
16                   :: go[i] ->
17                        go[i] = false; // stop!
18                   :: else -> skip;
19                   fi
20               :: else ->
21                   if
22                   :: go[i] ->
23                        busy++;
24                   :: else -> skip;
25                   fi
26              fi }
27         task_shed ! false }
28   :: BAD -> break;
29   od;
30 }
```

Планировщики с приоритетом ближайшего дедлайна (EDF) при выдаче разрешений на выполнение запущенных работ используют обновляемую очередь работ que размера, равного числу задач в системе. Работы в очереди располагаются в порядке EDF ближайшего дедлайна: чем ближе дедлайн работы задачи i, тем ближе она к голове очереди, то есть тем меньше номер элемента que, который хранит значение i. Для работы с очередями Promela модель всей системы реального времени содержит `inline`-вставки:

- `insert_queEDF(i)` для добавления работы задачи *i* в очередь согласно порядку EDF;
- `delete_que(i)` для удаления работы задачи *i* из очереди;
- `rearrange_que()` для сжатия очереди после удаления работ.

Promela-код этих вставок можно найти в репозитории[3].

Рассмотрим Promela-модель невытесняющего планировщика с приоритетом ближайшего дедлайна (NP-EDF) на листинге 7. В строках 5–11 планировщик, обнаружив только что запущенную работу (строка 7), помещает её в очередь на выполнение согласно приоритету EDF (строка 8). После подсчёта количества свободных процессоров (строка 12), планировщик последовательно просматривает очередь ожидающих работ (строка 13) и выдает разрешение на выполнение в порядке очередности (строка 17) до исчерпания количества свободных процессоров или очереди ожидающих работ (строка 15). При выдаче разрешения на выполнение, работа удаляется из очереди в строке 18 путём присваивания соответствующему элементу массива que значения 255, играющего роль значения *nil*. После удаления из очереди элементов следует все значения *nil* поместить после значимых значений. Эта операция выполняется в `rearrange_que()`.

Листинг 7: Promela-процесс для планировщика NP-EDF

```
1  proctype schedulerNPEDF(){
2   do
3   :: task_shed ? true ->
4   atomic{
5    for (i : 0 .. NumTask - 1){
6     if
7     :: release[i] && !go[i] ->
8              insert_queEDF(i)
9     :: else -> skip;
10    fi
11   }
12   free = NumProc - busy;
13   for (i : 0 .. NumTask - 1){
14    if
15    :: free == 0 || que[i] == 255 -> break;
16    :: else ->
17      go[que[i]] = true; // go!
18      que[i] = 255;
19      busy++;
20      free--;
21    fi
22   }
23   rearrange_que()
24   task_shed ! false
25  }
26  :: BAD -> break;
27  od;
28 }
```

Код вытесняющего планировщика с приоритетом ближайшего дедлайна (P-EDF) представлен на листинге 8. В строках 5–13 планировщик обрабатывает события завершения работ. Любая из выполняющихся работ может быть вытеснена, следовательно алгоритм этого планировщика должен перестраивать очередь работ при каждом их запуске или завершении. Поэтому, кроме запусков и ожидающих в очереди работ, соответствующих для работы `i` предикату `release[i]&&!go[i]`, этот планировщик также должен явно учитывать только что завершившиеся работы `end[i]` и удалять их из очереди в строке 8. Это явное определение только завершившийся работы необходимо, чтобы при удалении из очереди отличить ее от не запущенной работы, так как для той и другой значения в массивах `release[i]` и `go[i]` совпадают и равны *false*. После удаления работ из очереди

[3] https://github.com/GaraninaN/RealTimeSystems/blob/main/rts.pml

`rearrange_que()` приводит очередь в порядок (строка 13). Далее планировщик P-EDF, как планировщик NP-EDF записывает незапущенные работы в очередь (строка 17) и, как планировщик P-GFP, выдаёт разрешения всем работам в очереди (строка 18), чтобы далее отозвать лишние (строки 23–42).

Листинг 8: Promela-процесс для планировщика P-EDF

```
1  proctype schedulerPEDF(){
2   do
3   :: task_shed ? true ->
4         atomic{
5           for (i : 0 .. NumTask - 1){
6                if
7                :: end[i] ->
8                     delete_que(i);
9                     end[i] = false;
10               :: else -> skip;
11               fi
12          }
13          rearrange_que()
14          for (i : 0 .. NumTask - 1){
15               if
16               :: release[i] && !go[i] ->
17                    insert_queEDF(i)
18                    go[i] = true;
19               :: else -> skip;
20               fi
21          }
22          busy = 0;
23          for (i : 0 .. NumTask - 1){
24               if
25               :: que[i] == 255 -> break;
26               :: else -> skip;
27               fi
28               if
29               :: busy == NumProc ->
30                    if
31                    :: go[que[i]] ->
32                         go[que[i]] = false; // stop!
33                    :: else -> skip;
34                    fi
35               :: else ->
36                    if
37                    :: go[que[i]] ->
38                         busy++;
39                    :: else -> skip;
40                    fi
41               fi
42          }
43          task_shed ! false
44       }
45   :: BAD -> break;
46   od;
47  }
```

Код последних двух планировщиков может быть легко адаптирован для планировщиков, использующих другие приоритеты: для этого достаточно реализовать на Promela `inline`-вставку `insert_quePRTY(i)` для выбранного приоритета PRTY и поместить её «вызов» в строки 8 или 17 листингов 7 или 8 соответственно. Полная модель Promela для систем реального времени с планировщиками NP-GFP, P-GFP, NP-EDF и P-EDF находится в репозитории[4].

[4] https://github.com/GaraninaN/RealTimeSystems/blob/main/rts.pml

Table 1. Time consumption for systems with different schedulers and different numbers of tasks and processors

Таблица 1. Временные затраты анализа систем с различными планировщиками и различным числом задач и процессоров

	1	2	3	4	5	6	7
Задачи Процессоры	n = 5, m = 3	n = 6, m = 3	n = 6, m = 2	n = 7, m = 3	n = 7, m = 2	n = 7, m = 4	n = 10, m = 4
NPGPF	0,817	24,5	0,009	1,54		0	0,013
PGPF	0,9	11	0,002	0,012		0	0,004
NPEDF	1,04	0	0,002	11		0	0,047
PEDF	0,879	0	0,002	0	0,011	0	0,015
Utilization	58	68	109	87		65	98/4 79/5

	8	9	10	11	12	13	14
Задачи Процессоры	n = 20, m = 9	n = 40, m = 20	n = 60, m = 31	n = 80, m = 42	n = 80, m = 41	n = 100, m = 53	n = 120, m = 64
NPGPF	0,006	0,28	0,059	0,098		0,132	0,185
PGPF	0,01	0,041	0,079	0,144		0,203	0,303
NPEDF	6,77	0,072	0,181	0,349		0,612	1,01
PEDF	0,04	0,346	0,837	0	1,08	2,9	3,17
Utilization	95/9 85/10	90/20 86/21	89/31 87/32	89/42 87/43		89/53 87/54	89/64 88/65

3. Эксперименты

Чтобы точно протестировать систему реального времени на возможность планирования с выбранным планировщиком, мы проверяем выполнимость формулы LTL `[]!BAD` в Promela-модели этой системы с помощью инструмента верификации моделей SPIN. Если эта формула выполнима, ни одна задача проверяемой системы реального времени не пропускает свой дедлайн. Мы провели ряд экспериментов, используя версию 6.5.1 верификатора SPIN на ноутбуке с ЦП 11 Gen Intel(R) Core(TM) i3-1115G4@3.00GHz 3.00 GHz с 4 ядрами и 12 ГБ ОЗУ.

Чтобы проверить работоспособность нашего подхода, мы использовали синтетические данные для параметров задач системы реального времени: каждая задача с номером i имела время выполнения $C_i = i + 1$ и относительный дедлайн $D_i = 2 \cdot (i + 2)$, равный периоду P_i. В ходе экспериментов мы определили:

- максимальное число задач и процессоров для которых анализ безопасной системы заканчивается за приемлемое время для всех планировщиков (колонка 1, таблица 1);
- максимальное число задач и процессоров для которых анализ безопасной системы заканчивается за приемлемое время для одного планировщика (колонка 2, таблица 1);
- минимальное число задач и процессоров для которых анализ системы не заканчивается за приемлемое время для всех планировщиков (колонка 6, таблица 1);
- для числа задач выше 10 — максимальное число процессоров для которых анализ системы не заканчивается за приемлемое время для всех планировщиков, а при уменьшении числа процессоров на 1 верификатор признаёт систему небезопасной (колонки 8–14, таблица 1).

Кроме того, мы подсчитали значение утилизации для каждой системы. В колонках с 7 по 14 приведены значения утилизации для заглавного числа процессоров и для числа процессоров на 1 меньше: например, в колонке 7 для 10 задач и 4 процессоров значение утилизации равно 98, а для 10 задач и 5 процессоров значение утилизации равно 79.

Результаты экспериментов приведены в таблице 1. Первая колонка таблицы содержит названия планировщиков. Столбцы помечены количеством задач n и количеством процессоров m. В ячейках

таблицы приведено время работы SPIN в секундах. Зелёные ячейки отмечают случаи, когда доказано, что система безопасна, то есть верификатор SPIN подтвердил, что плохие состояния системы, в которых какая-либо задача пропускает свой дедлайн, не достигаются. Красные ячейки помечают случаи, когда доказано, что система небезопасна, то есть верификатор SPIN выработал контрпример — последовательность запусков работ, приводящую к состоянию, в котором какая-либо задача пропускает свой дедлайн, но если увеличить число процессоров на 1 (понизить значение утилизации), то SPIN в течение долгого времени (более часа) контрпример не находит. Синие ячейки помечают случаи, когда верификатор работал относительно долго (более часа), но свою работу не завершил, то есть не вынес вердикт о безопасности или небезопасности Promela-модели системы реального времени.

Таблица показывает, что на данной аппаратуре SPIN может доказать безопасность системы для 5 задач и 3 процессоров для всех планировщиков (колонка 1), и для 6 задач и 3 процессоров для невытесняющего планировщика с глобальным фиксированным приоритетом (колонка 2). При уменьшении количества процессоров или увеличении количества задач система становится доказуемо небезопасной (колонки 3, 4) с большинством планировщиков. При увеличении числа и задач, и процессоров, верификатор SPIN перестает справляться с доказательствами безопасности (колонка 7), поэтому для большего числа процессоров и задач эксперименты в контексте доказательства безопасности оказываются бессмысленными. При этом небезопасность системы была доказана для максимального числа задач, которое можно было определить в разработанной выше Promela-модели (колонки 7–14).

С одной стороны, увеличение количества процессоров для одной и той же системы задач понижает значение утилизации, которое соответствует загруженности процессоров, поэтому большее количество процессоров может сделать небезопасную систему безопасной[5]. С другой стороны, имеет место быстрота обнаружения небезопасности системы реального времени (обычно доли секунд или секунды) и долгое время доказательства её безопасности (секунды). Эти наблюдения подсказывают следующие практические соображения для проведения теста на планируемость с помощью метода проверки моделей и верификатора SPIN. Если система с n задачами и m процессорами *доказуемо небезопасна*, а безопасность/небезопасность тех же задач на $m + 1$ процессорах *не доказана* в течение заданного срока (более одного часа), то практически можно считать, что последняя система *безопасна*. Следуя этому правилу, можно считать, что для 6 задач и 3 процессоров планировщики ближайшего дедлайна EDF, как вытесняющий, так и невытесняющий, скорее всего, делают систему реального времени безопасной, в отличие от вытесняющего планировщика с фиксированным приоритетом (колонка 3), система с 7 задачами и 4 процессорами безопасна со всеми планировщиками (колонка 6), а вытесняющий планировщик EDF в двух случаях более безопасен, чем все остальные (колонки 4 и 11). Отношения долей времени здесь приведены для аппаратуры и размеров задач, использовавшихся в проведенных экспериментах. Для других параметров возможны другие соотношения. Например, для больших систем реального времени и аппаратуры с большим объемом оперативной памяти могут затрачиваться секунды для доказательства небезопасности, минуты для доказательства безопасности, и отсутствием доказательства можно считать ситуацию, когда верификатор не выносит вердикт в течении нескольких часов.

Сравним производительность нашего подхода и существующих в настоящее время подходов к точной оценке безопасности (планируемости) систем реального времени. В таблице 2 представлены как методы проверки общего назначения (guan07 [13], suk15 [16], yal19 [14], our — данная работа), так и специализированные методы (sun16 [8], geer17 [17], bur22 [5], ran23 [6]). Работы guan07 [13]

[5]Утилизация принимает минимальное значение для заданного набора задач, когда количество процессоров равно количеству задач. В этом случае очевидно, что любая система с задачами, в которых время исполнения меньше относительного дедлайна является безопасной.

Table 2. Comparison of the performance of our approach and state-of-the-art approaches

Таблица 2. Сравнение производительности нашего подхода и существующих подходов

	guan07	suk15	sun16	geer17	yal19	bur22	ran23	our
Задачи	6	10	5	4	12	8	7	6
Проц.	5	1	2	3	8	4	1	5
Время	400 sec	4.44 sec	400 min	2200 sec	1000 sec	5000 sec	0.005 sec	22 sec
Память	700 Mb	439	$6{\cdot}10^4$	$43{\cdot}10^4$	–	$10{\cdot}10^7$	766	$12{\cdot}10^6$
Метод	UPPAAL	SPIN	LHA	BWR	UPPAAL	Pruning	POD	SPIN

и yal19 [14] используют представление систем реального времени как временных автоматов и соответствующий инструмент проверки моделей UPPAAL. Наша работа и работа suk15 [16] описывают системы реального времени на языке Promela и проверяют их безопасность в инструмента SPIN. Работа sun16 [8] предлагает моделирование систем с помощью линейных гибридных автоматов, а geer17 [17] использует технику обратной достижимости в играх на графах. В работе bur22 [5] развивается метод отсечения ветвей обхода графа достижимых состояний, основанный на достаточных условиях недостижимости плохих состояний, ran23 [6] напрямую применяет метод редукции частичных порядков для снижения числа проверяемых путей, используемый для тех же целей в SPIN. Данные таблицы взяты из разделов упомянутых статей, посвященных экспериментальным исследованиям. В таблице приведено использование памяти и времени для максимального числа задач и процессоров из экспериментов, поскольку в одной из статей было замечено, что на ресурсоёмкость доказательства безопасности систем реального времени больше влияет количество задач и процессоров, чем значение утилизации. Единицы измерения времени указаны рядом с их значениями. Единицы измерения памяти везде, кроме первого столбца — это количество состояний, исследованных в процессе доказательства. К сожалению, не везде эти данные приведены. Нужно отметить, что точные тесты проводились для разных планировщиков и систем задач, включая приостанавливающиеся задачи и задачи с колебаниями параметров. При этом экспериментальные исследования производились на различной аппаратуре, как на обычных десктопных устройствах и даже ноутбуках, как в данной работе, так и на кластерах Intel Xeon. По этим причинам реальное сравнение производительности затруднительно, однако приведённая таблица 2 показывает, что точные методы определения безопасности систем реального времени в целом являются довольно ресурсоемкими, но при этом предложенной в настоящей работе подход оказывается достаточно конкурентоспособным с учётом того, что он позволяет одну и ту же систему реального времени проверять для различных планировщиков без переписывания её модели.

В итоге, эксперименты показывают, что инструмент SPIN хорошо подходит для доказательства небезопасности систем реального времени с различными планировщиками. Если принять соображения, изложенные выше, то на практике невозможность доказать небезопасность системы в течение заданного времени может служить критерием доверия к ее безопасности.

Заключение

В этой статье мы формализуем системы реального времени с абстрактным планировщиком как модели Крипке реального времени. Мы показываем, что абстрактный планировщик может быть уточнен в терминах предложенной модели. Определив условия запуска работ задач с помощью переменных, значения которых задают состояния модели, мы представили невытесняющий планировщик с глобальным фиксированным приоритетом, вытесняющий планировщик с глобальным фиксированным приоритетом, невытесняющий планировщик с приоритетом ближайшего дедлайна и вытесняющий планировщик с приоритетом ближайшего дедлайна. Этот формализм

можно использовать также для определения динамических онлайн планировщиков с другими способами задания приоритетов.

Мы также реализовали предложенную модель Крипке реального времени на языке Promela верификатора моделей SPIN для решения основной задачи систем реального времени — проведения точного теста на планируемость системы с помощью выбранного планировщика. Проведенные эксперименты показали, что инструмент SPIN хорошо подходит для доказательства небезопасности систем и, при некоторых условиях, для проверки их безопасности. Кроме проведения точных тестов планируемости, предложенная модель может успешно использоваться в учебных курсах по системам реального времени в силу простоты её адаптации для различных параметров систем и планировщиков. Кроме проверки свойства безопасности (планируемости), адаптированную Promela-модель можно применять в задаче построения расписаний для офлайн планировщиков на основе часов, используя метод контрпримеров.

В будущем мы планируем использовать нашу формализацию систем реального времени для разработки новых эффективных алгоритмов для точного теста планируемости, например, алгоритмов на основе обратного прослеживания.

References

[1] J. W. S. Liu, *Real-time systems*. Prentice-Hall, 2001.

[2] B. B. Brandenburg and M. Gül, “Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations”, in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 99–110. DOI: 10.1109/RTSS.2016.019.

[3] Q. Zhou, G. Li, C. Zhou, and J. Li, “Limited busy periods in response time analysis for tasks under global EDF scheduling”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 2, pp. 232–245, 2021. DOI: 10.1109/TCAD.2020.2994265.

[4] A. Burmyakov, E. Bini, and E. Tovar, “An exact schedulability test for global FP using state space pruning”, in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, 2015, pp. 225–234. DOI: 10.1145/2834848.2834877.

[5] A. Burmyakov, E. Bini, and C.-G. Lee, “Towards a tractable exact test for global multiprocessor fixed priority scheduling”, *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2955–2967, 2022. DOI: 10.1109/TC.2022.3142540.

[6] S. Ranjha, P. Gohari, G. Nelissen, and M. Nasri, “Partial-order reduction in reachability-based response-time analyses of limited-preemptive DAG tasks”, *Real-Time Systems*, vol. 59, no. 2, pp. 201–255, 2023. DOI: 10.1007/s11241-023-09398-x.

[7] P. Gohari, J. Voeten, and M. Nasri, “Reachability-based response-time analysis of preemptive tasks under global scheduling”, in *Proceedings of the 36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, 2024, 3:1–3:24. DOI: 10.4230/LIPIcs.ECRTS.2024.3.

[8] Y. Sun and G. Lipari, “A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling”, *Real-Time Systems*, vol. 52, no. 3, pp. 323–355, 2016. DOI: 10.1007/s11241-015-9245-9.

[9] A. M. K. Cheng, *Real-Time Systems: Scheduling, Analysis, and Verification*. John Wiley & Sons, 2002.

[10] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, *et al.*, *Handbook of model checking*. Springer, 2018, vol. 10.

[11] G. J. Holzmann, *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003, 608 pp.

[12] R. Cavada *et al.*, "The nuXmv symbolic model checker", in *Computer Aided Verification*, 2014, pp. 334–342. DOI: 10.1007/978-3-319-08867-9_22.

[13] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu, "Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking", in *Software Technologies for Embedded and Ubiquitous Systems*, 2007, pp. 263–272.

[14] B. Yalcinkaya, M. Nasri, and B. B. Brandenburg, "An exact schedulability test for non-preemptive self-suspending real-time tasks", in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1228–1233. DOI: 10.23919/DATE.2019.8715111.

[15] S. M. Staroletov, "A formal model of a partitioned real-time operating system in Promela", *Proceedings of the Institute for System Programming of the RAS*, vol. 32, no. 6, pp. 49–65, 2020, in Russian.

[16] Sukvanich, Punwess, Thongtak, Arthit, and Vatanawood, Wiwat, "Formalizing real-time embedded system into Promela", *MATEC Web of Conferences*, vol. 35, p. 03 003, 2015. DOI: 10.1051/matecconf/20153503003.

[17] G. Geeraerts, J. Goossens, and T.-V.-A. Nguyen, "A backward algorithm for the multiprocessor online feasibility of sporadic tasks", in *Proceedings of the 17th International Conference on Application of Concurrency to System Design (ACSD)*, 2017, pp. 116–125. DOI: 10.1109/ACSD.2017.9.

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[19] V. Bonifaci and A. Marchetti-Spaccamela, "Feasibility analysis of sporadic real-time multiprocessor task systems", *Algorithmica*, vol. 63, pp. 763–780, 2010.

[20] C. A. R. Hoare, "Communicating sequential processes", *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[21] N. O. Garanina, "An exact schedulability test for real-time systems with an abstract scheduler", *System Informatics*, vol. 25, pp. 29–38, 2024. DOI: 10.31144/si.2307-6410.2024.n25.p29-38.