MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 33, NO. 1, 2026
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Using Symmetry in Programming and Verification of a Resource Arbiter

M. V. Neyzov[1]

DOI: 10.18255/1818-1015-2026-1-90-116

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

MSC2020: 68Q60
Research article
Full text in Russian

Received December 29, 2025
Revised February 24, 2026
Accepted February 27, 2026

The large state space of programs makes their direct verification by model checking difficult or impossible. The presence of symmetry in a program often allows simplifying the model and reducing its state space, leading to significant decrease of verification time. The classical approach consists in detecting a symmetry group and constructing a quotient model based on it — a simplified model for verification purposes. However, not all tools provide support for symmetry, and those that do may still struggle because finding an appropriate symmetry group is computationally complex problem.

This work proposes an approach to program development based on explicit symmetry exploitation, which is an alternative to the classical one. In the program, a core is extracted — a coordination center working under consideration of symmetry and responsible for ensuring temporal properties. The core coordinates computations outside itself — those placed in the wrapper surrounding the core. As a result, the core has a small state space, replace the quotient model and allows verification using a model checker without symmetry support. The wrapper cannot interfere in the operation of the verified core and violate its properties. The approach is demonstrated by the example of the development and verification of the Mars rover resource arbiter. The arbiter coordinates access of n processes to m resources where both n and m are natural numbers. Programming languages C/C++ and the Spin model checker tool are used. The behavioral model of the core is automatically extracted by the Spin tool from the C code. Temporal properties expressed via Linear Temporal Logic (LTL) are subject to verification.

Keywords: temporal properties; model checking; model extracting; Spin model checker; C++ language; transition system; program core; index automaton; simulation; bisimulation; symmetry group; quotient model; state orbit; arbiter of rover resources

INFORMATION ABOUT THE AUTHORS

Neyzov, Maxim V. (corresponding author) | ORCID iD: 0009-0000-6893-6137. E-mail: neyzov.max@gmail.com
Researcher

Funding: IAaE SB RAS (Project No 125022803031-1).

For citation: M. V. Neyzov, “Using symmetry in programming and verification of a resource arbiter”, *Modeling and Analysis of Information Systems*, vol. 33, no. 1, pp. 90–116, 2026. DOI: 10.18255/1818-1015-2026-1-90-116.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 33, № 1, 2026
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Использование симметрии при программировании и верификации арбитра ресурсов

М. В. Нейзов[1]

DOI: 10.18255/1818-1015-2026-1-90-116

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

УДК 004.41+519.683
Научная статья
Полный текст на русском языке

Получена 29 декабря 2025 г.
После доработки 24 февраля 2026 г.
Принята к публикации 27 февраля 2026 г.

Большое пространство состояний программ затрудняет или делает невозможной их непосредственную верификацию методом проверки модели (model checking). Наличие симметрии в программе достаточно часто позволяет упростить модель и сократить её пространство состояний, что приводит к значительному снижению времени верификации. Классический подход состоит в обнаружении группы симметрии и построении на её основе фактормодели — упрощённой модели для верификации. Однако не все инструменты имеют поддержку симметрии, а инструменты с такой поддержкой не всегда справляются с поставленной задачей, так как поиск подходящей группы симметрии является вычислительно сложной задачей.

В работе предлагается альтернативный классическому подход к разработке программ, основанный на явном выделении симметрии. В программе выделяется ядро — координационный центр, который работает с учётом симметрии и отвечает за выполнимость заданных темпоральных свойств. Ядро координирует вычисления, вынесенные за его пределы — в обвязку ядра. В связи с этим ядро имеет небольшое пространство состояний, заменяет собой фактормодель и может быть верифицировано инструментом проверки модели без поддержки симметрии. Обвязка не может вмешиваться в работу верифицированного ядра и нарушать его свойства. Подход продемонстрирован на примере разработки и верификации арбитра ресурсов марсохода. Арбитр координирует доступ n процессов к m ресурсам, где n и m — натуральные числа. Используются язык программирования C/C++ и инструмент проверки модели Spin. Модель поведения ядра автоматически извлекается верификатором Spin из C-кода. Проверке подлежат темпоральные свойства, выраженные с помощью линейной темпоральной логики LTL.

Ключевые слова: темпоральные свойства; проверка модели; извлечение модели; верификатор Spin; язык C++; система переходов; ядро программы; индексный автомат; симуляция; бисимуляция; группа симметрии; фактормодель; орбита состояния; арбитр ресурсов марсохода

ИНФОРМАЦИЯ ОБ АВТОРАХ

Нейзов, Максим Вячеславович (автор для корреспонденции) | ORCID iD: 0009-0000-6893-6137. E-mail: neyzov.max@gmail.com
Исследователь

Финансирование: ИАиЭ СО РАН (проект № 125022803031-1).

Для цитирования: M. V. Neyzov, “Using symmetry in programming and verification of a resource arbiter”, *Modeling and Analysis of Information Systems*, vol. 33, no. 1, pp. 90–116, 2026. DOI: 10.18255/1818-1015-2026-1-90-116.

Введение

Программные *реагирующие системы* [1] постоянно взаимодействуют со своим окружением, работают циклически и потенциально никогда не завершаются. Основное назначение таких систем — обеспечить ожидаемое окружением поведение, которое можно представить в виде набора свойств, выраженных с помощью некоторой *темпоральной логики* [2].

Проверка соответствия программы заданным свойствам называется *верификацией* [3]. Подходящей и широко используемой технологией формальной верификации реагирующих систем является *проверка модели* (model checking) [4—6]. При этом верификации подвергается не сама программа, а её модель, представляющая собой конечную *систему переходов* [4—6], которая отражает все возможные переходы между состояниями программы.

Наличие *симметрии* в системе переходов означает, что существует нетривиальная *группа перестановок* на множестве состояний, сохраняющая выполнимость проверяемых темпоральных свойств [5]. На основе данной *группы симметрии* строится *фактормодель* — редуцированная (упрощённая) модель поведения программы, отражающая классы эквивалентных переходов.

Реальные программы и соответствующие им модели поведения обычно имеют большое число состояний, поэтому их непосредственная верификация затруднена [7]. Однако при наличии симметрии возможна эффективная редукция модели, которая сокращает число состояний и делает верификацию приемлемой по затрачиваемым вычислительным ресурсам (времени и памяти) [8, 9]. Согласно данному подходу для упрощения верификации требуется [5, 9]:

1. Выявить имеющуюся группу симметрии в исходной модели поведения программы.
2. С помощью данной группы симметрии построить редуцированную модель (фактормодель).

Результаты проверки свойств редуцированной модели могут быть перенесены на исходную модель и программу соответственно.

Не все инструменты проверки модели имеют поддержку симметрии, т. е. не позволяют выполнять действия 1 и 2 в автоматическом или автоматизированном режиме. Инструменты с поддержкой симметрии не всегда удовлетворительно справляются с поставленной задачей.

Часто уже на этапе разработки известна присущая программе симметрия. Более того, описывая детали поведения программы, разработчик сам закладывает в неё конкретный вид симметрии, только делает это неявным образом для себя и инструментов верификации.

В настоящей работе предлагается подход к разработке программ, основанный на явном отражении в них симметрии. Для этого в программе выделяется ядро — координационный центр, отвечающий за выполнимость заданных темпоральных свойств. Ядро проектируется с учётом симметрии: не отличает эквивалентные состояния программы и переходы между ними. Для этого выполняется абстрагирование от конкретных значений параметров реагирующей системы. Работа ядра индуцирует некоторую группу симметрии. При этом разработчику даже не важно знать какую именно группу симметрии задаёт разработанное им ядро, главное, чтобы оно обеспечивало выполнимость заданных темпоральных свойств. Модель поведения ядра заменяет собой фактормодель, извлекается из кода программы без каких-либо абстракций и подвергается верификации методом проверки модели.

Остальная часть программы — обвязка ядра, которая отвечает за различные вычисления, выполняемые под управлением ядра. Здесь осуществляется хранение и вычисление значений всех параметров реагирующей системы. Обвязка имеет большое пространство состояний, так как решает вычислительную задачу. Ядро же, наоборот, содержит небольшое число состояний, так как выполняет только управляющую/координационную функцию. Обвязка не может вмешаться в работу ядра и нарушить темпоральные свойства.

В работе используются языки программирования C/C++ и инструмент проверки модели Spin, который позволяет выполнять автоматическое извлечение модели из C-кода. Таким образом, пред-

ложенный в статье подход избавляет разработчика от выполнения действий 1 и 2 (см. выше) при верификации на основе фактормодели. Демонстрация подхода выполняется на примере разработки и верификации арбитра ресурсов марсохода.

Содержание работы. Раздел 1 содержит обзор связанных работ. В разделе 2 вводятся формальные определения модели и фактормодели поведения программы, симуляции и бисимуляции, темпоральной логики LTL и группы симметрии. В разделе 3 выполняется постановка задачи разработки арбитра ресурсов, в разделе 4 приводится её решение. Раздел 5 показывает группу симметрии разработанной программы. Для этого устанавливается связь между моделями поведения арбитра и ядра. В разделе 6 выполняется верификация ядра арбитра. В заключительном разделе подводятся итоги работы.

1. Обзор связанных работ

Обзор работ о применении симметрии для проверки моделей темпоральных свойств представлен в [10]. При наличии идентичных компонентов в параллельных системах удаётся добиться экспоненциального снижения сложности проверки моделей [11]. Использование симметрии упрощает верификацию для инструментов, работающих на основе BDD [12] и SAT [13]. Проблема обнаружения симметрии (выявления орбит) является вычислительно сложной задачей [8—10].

Для упрощения выявления симметрии при верификации применяются специальным образом аннотированные модели [14]. В [15] для упрощения обнаружения симметрии в язык описания вводится специальный тип данных *scalarset* — конечное неупорядоченное множество, перестановка элементов которого не влияет на проверяемые свойства. С учётом этого верификатор Mur генерирует сокращённое пространство состояний. Инструмент символьной проверки модели SYMM позволяет пользователю задавать используемую при верификации CTL-формул симметрию [8].

В [16] предлагается индуктивный подход к выводу симметрий в параметризованных системах, в [8] — методы обнаружения симметрии для параллельных программ с разделяемыми переменными. Для B и Z моделей представлена эффективная приближенная верификация на основе вычисляемых для состояний маркеров симметрии [17].

Инструмент проверки модели SMC для верификации темпоральных свойств строит сокращённый граф состояний на основе симметрии процессов [18]. В [19] исследованы специальные GQS-структуры (англ. Guarded Annotated Quotient Structures) для проверки темпоральных свойств, которые различают симметричные состояния. Также GQS-структуры позволяют компактно представлять поведение систем со слабой симметрией и без неё. Алгоритм проверки свойств на GQS-структурах является расширением для инструмента SMC.

Инструмент SymmExtractor получает на вход спецификацию на языке Promela и выполняет автоматическое обнаружение и вычисление группы симметрии, обусловленной коммуникациями в каналах [20]. Синхронизация, введенная на уровне языка SPSL, используется для редукции модели, состоящей из симметричных взаимодействующих модулей [21]. Для инструмента Spin разработан пакет редукции моделей с произвольной группой структурных симметрий [22]. В [23] для верификации моделей, представленных в виде взаимодействующих последовательных процессов (Communicating Sequential Processes, CSP), представлено расширение инструмента проверки модели FDR, которое использует симметрию для сокращения пространства состояний.

Для моделей, выраженных на языке Rebeca, представлены два подхода к обнаружению симметрии: первый подход основан на топологии взаимосвязей между объектами, второй — на структурах данных [24]. Для средства проверки моделей ProB реализована редукция модели на основе симметрии с помощью переполнения перестановками (англ. permutation flooding) [25].

Таким образом, рассмотренные выше работы сосредоточены на обнаружении симметрии и построении фактормодели на её основе. Это требует специальной инструментальной поддержки. Предлагаемый в настоящей работе подход ориентирован на разработку ядра, которое по сути

уже является редуцированной моделью. Это позволяет использовать любые инструменты проверки модели при верификации.

Симметричные свойства алгоритма навигации роя роботов использовались для упрощения верификации [26]. Симметрия, возникающая за счёт резервирования компонентов системы контроля, использовалась при верификации функций защиты атомной электростанции [27, 28]. Верификация выполнялась с помощью инструмента проверки модели NuSMV [26—28]. В данных работах предлагается использовать знания о симметрии при разработке модели для верификации, в настоящей работе — при разработке ядра программы, которое также будет подвергаться верификации.

В [29] предложена разработка параллельных программ на основе каркасов синхронизации (англ. synchronization skeletons), которые извлекаются из синтезированной конечной модели CTL-формулы, описывающей требуемое поведение программы. Предлагаемое в настоящей работе ядро похоже на каркас синхронизации: ядро координирует работу всех внешних по отношении к нему компонентов программы, каркас синхронизирует параллельные процессы, входящие в состав программы. Цель ядра и каркаса — обеспечить выполнимость заданных темпоральных свойств. Ядро, в отличии от каркаса, проектируется разработчиком и учитывает присущую программе симметрию.

2. Предварительные сведения

Система переходов (англ. transition system) [4—6] представляет собой формальную модель поведения программы. Система переходов $TS = \langle S, S_0, R, P, L\rangle$, где S — множество состояний (каждое состояние программы представляет собой вектор зафиксированных в определённый момент времени значений переменных), $S_0 \subseteq S$ — множество начальных состояний, $R \subseteq S \times S$ — *тотальное* отношение переходов. Свойство тотальности имеет вид $(\forall s \in S)\,(\exists s' \in S)\,(s, s') \in R$, т. е. существует переход из любого состояния. Каждый цикл работы программы приводит к переходу. Если после цикла работы значения переменных программы не изменились, то выполнился переход по петле в то же самое состояние. $P = \{p_1, \ldots, p_m\}$ — множество атомарных утверждений относительно значений переменных, $L\colon S \to 2^P$ — функция разметки состояний атомарными утверждениями, истинными в этих состояниях.

Путь $\pi = s_0\, s_1\, s_2 \ldots \in S^\omega$ в системе переходов TS представляет собой сценарий поведения программы, где S^ω — множество всех бесконечных слов в алфавите S. Сама система переходов TS задаёт множество путей Π_{TS}, которые начинаются в начальном состоянии:

$$\Pi_{TS} = \{\, \pi \in S^\omega \mid \big(\pi(0) \in S_0\big) \wedge (\forall i \in \mathbb{N}_0)\big(\pi(i), \pi(i+1)\big) \in R \,\},$$

где $\pi(i) = s_i$ — i-ое состояние пути π, $i \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

Обозначим π^i суффикс пути π, который начинается с состояния $\pi(i)$, т. е. $\pi^i = s_i\, s_{i+1}\, s_{i+2} \ldots \in S^\omega$.

Логика LTL. Формализация требований к программе выполняется с помощью *линейной темпоральной* логики (Linear Temporal Logic, LTL) [4, 5]. Для атомарных утверждений $p \in P$ синтаксис LTL-формул имеет вид:

$$\varphi, \psi ::= \mathit{true} \mid \mathit{false} \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \psi\mathbf{U}\varphi \mid \psi\mathbf{W}\varphi.$$

Отношение *выполнимости* $\models$ LTL-формулы для пути π имеет вид:

$$\begin{array}{ll}
\pi \models \mathit{true}; & \pi \not\models \mathit{false};\\
\pi \models p & \Longleftrightarrow p \in L\big(\pi(0)\big);\\
\pi \models \neg\varphi & \Longleftrightarrow \pi \not\models \varphi;\\
\pi \models \varphi \wedge \psi & \Longleftrightarrow \pi \models \varphi \ \text{ и } \ \pi \models \psi;\\
\pi \models \varphi \vee \psi & \Longleftrightarrow \pi \models \varphi \ \text{ или } \ \pi \models \psi;\\
\pi \models \varphi \Rightarrow \psi & \Longleftrightarrow \pi \models \neg\varphi \ \text{ или } \ \pi \models \psi;
\end{array}$$

$$
\begin{array}{ll}
\pi \models \mathbf{X}\varphi & \Longleftrightarrow \pi^1 \models \varphi; \\
\pi \models \mathbf{F}\varphi & \Longleftrightarrow (\exists i \geq 0)\ \pi^i \models \varphi; \\
\pi \models \mathbf{G}\varphi & \Longleftrightarrow (\forall i \geq 0)\ \pi^i \models \varphi; \\
\pi \models \psi \mathbf{U} \varphi & \Longleftrightarrow (\exists i \geq 0)\ \pi^i \models \varphi \ \text{ и } \ (\forall j,\ 0 \leq j < i)\ \pi^j \models \psi; \\
\pi \models \psi \mathbf{W} \varphi & \Longleftrightarrow (\psi \mathbf{U} \varphi) \ \text{ или } \ \mathbf{G}(\psi).
\end{array}
$$

Данные темпоральные операторы читаются следующим образом: $\mathbf{X}(\varphi)$ — в следующий момент времени верно φ, $\mathbf{F}(\varphi)$ — в будущем верно φ, $\mathbf{G}(\varphi)$ — всегда φ, $\psi \mathbf{U} \varphi$ — в будущем φ, и пока оно не достигнуто должно быть верно ψ, $\mathbf{W}$ — слабая форма оператора $\mathbf{U}$, т. е. не требуется обязательное появление φ в будущем.

Отношение выполнимости $\models$ LTL-формулы для системы переходов имеет вид:

$$TS \models \varphi \Longleftrightarrow (\forall \pi \in \Pi_{TS})\ \pi \models \varphi.$$

Симуляция. Для систем переходов $TS = \langle S, S_0, R, P, L \rangle$ и $TS' = \langle S', S'_0, R', P', L' \rangle$ бинарное отношение $H \subseteq S \times S'$ является отношением *симуляции*, если [4, 5]:

$$(\forall s \in S)\ (\forall s' \in S')\ [H(s, s') \Rightarrow L(s) \cap P' = L'(s')], \tag{1}$$

$$(\forall s_1, s_2 \in S)\ (\forall s'_1 \in S')\ [H(s_1, s'_1) \wedge R(s_1, s_2) \Rightarrow (\exists s'_2 \in S')\ R'(s'_1, s'_2) \wedge H(s_2, s'_2)], \tag{2}$$

т. е. разметка атомарными утверждениями из P' соответствующих состояний совпадает (1), каждому переходу в системе переходов TS соответствует переход в системе переходов TS' (2).

Система переходов TS' *симулирует* TS (обозначим $TS \preceq TS'$), если существует такое отношение симуляции H, что [5, 30]:

$$(\forall s_0 \in S_0)\ (\exists s'_0 \in S'_0)\ H(s_0, s'_0), \tag{3}$$

т. е. для любого начального состояния в TS найдётся (в соответствии с отношением симуляции H) начальное состояние в TS'.

Известно, что для любой ACTL* формулы φ верно [4, 5]:

$$(TS' \models \varphi) \Rightarrow (TS \models \varphi), \tag{4}$$

т. е. допустимо проверять выполнимость свойств на симуляции (корректной абстракции). Утверждение (4) верно для любой LTL-формулы, так как логика LTL является подмножеством логики ACTL* [5].

Бисимуляция. Для систем переходов $TS = \langle S, S_0, R, P, L \rangle$ и $TS' = \langle S', S'_0, R', P, L' \rangle$ бинарное отношение $B \subseteq S \times S'$ является отношением *бисимуляции*, если [4, 5, 30]:

$$(\forall s \in S)\ (\forall s' \in S')\ [B(s, s') \Rightarrow L(s) = L'(s')], \tag{5}$$

$$(\forall s_1, s_2 \in S)\ (\forall s'_1 \in S')\ [B(s_1, s'_1) \wedge R(s_1, s_2) \Rightarrow (\exists s'_2 \in S')\ R'(s'_1, s'_2) \wedge B(s_2, s'_2)], \tag{6}$$

$$(\forall s_1 \in S)\ (\forall s'_1, s'_2 \in S')\ [B(s_1, s'_1) \wedge R'(s'_1, s'_2) \Rightarrow (\exists s_2 \in S)\ R(s_1, s_2) \wedge B(s_2, s'_2)], \tag{7}$$

т. е. разметка атомарными утверждениями соответствующих состояний совпадает (5), каждому переходу в системе переходов TS соответствует переход в системе переходов TS' (6). Это же верно и в обратную сторону: любому переходу в TS' соответствует переход в TS (7).

Системы переходов TS и TS' *бисимуляционно эквивалентны* [5, 30] (обозначим $TS \equiv TS'$), если существует такое отношение бисимуляции B, что:

$$(\forall s_0 \in S_0)\ (\exists s'_0 \in S'_0)\ B(s_0, s'_0), \tag{8}$$

$$(\forall s'_0 \in S'_0)\ (\exists s_0 \in S_0)\ B(s_0, s'_0), \tag{9}$$

т. е. для любого начального состояния в TS найдётся (в соответствии с отношением бисимуляции B) начальное состояние в TS', и наоборот.

Для бисимуляционно эквивалентных систем переходов существует теорема [5], которая утверждает, что для любой формулы φ логики CTL* верно:

$$(TS \models \varphi) \Leftrightarrow (TS' \models \varphi), \tag{10}$$

т. е. выполнимость/нарушение свойства φ можно проверять на любой из двух систем переходов (TS или TS') без потери достоверности полученного результата. Данная теорема верна для любой LTL-формулы, так как логика LTL является подмножеством логики CTL* [5].

Группа перестановок $G = \langle \Sigma, \circ \rangle$, где $\Sigma = \{\sigma \colon S \to S\}$ — набор биективных отображений (преобразований) множества состояний S на себя, бинарная операция композиции $\circ \colon \Sigma \times \Sigma \to \Sigma$. Группа перестановок G обладает тремя свойствами [10]:

1. Ассоциативность: $(\forall \sigma_1, \sigma_2, \sigma_3 \in \Sigma)[(\sigma_1 \circ \sigma_2) \circ \sigma_3 = \sigma_1 \circ (\sigma_2 \circ \sigma_3)]$.
2. Наличие *нейтрального* элемента: $(\exists e \in \Sigma)(\forall \sigma \in \Sigma)[e \circ \sigma = \sigma \circ e = \sigma]$.
3. Наличие *обратного* элемента: $(\forall \sigma \in \Sigma)(\exists \sigma^{-1} \in \Sigma)[\sigma \circ \sigma^{-1} = \sigma^{-1} \circ \sigma = e]$.

Группа инвариантности [5, 9] системы переходов TS — группа перестановок, которая для любого атомарного утверждения $p \in P$ и любого преобразования $\sigma \in \Sigma$ сохраняет разметку состояний:

$$(\forall p \in P)\,(\forall \sigma \in \Sigma)\,(\forall s \in S)\,[p \in L(s) \Leftrightarrow p \in L\big(\sigma(s)\big)]. \tag{11}$$

Группа автоморфизмов. Преобразование $\sigma \in \Sigma$ является *автоморфизмом* для системы переходов TS тогда и только тогда, когда σ сохраняет отношение переходов R [5, 9]:

$$(\forall s_1, s_2 \in S)\,[(s_1, s_2) \in R \Rightarrow \big(\sigma(s_1), \sigma(s_2)\big) \in R\,]. \tag{12}$$

Если каждое преобразование $\sigma \in \Sigma$ из группы перестановок $G = \langle \Sigma, \circ \rangle$ является автоморфизмом, то G — *группа автоморфизмов* [5, 9] для TS:

$$(\forall \sigma \in \Sigma)\,(\forall s_1, s_2 \in S)\,[(s_1, s_2) \in R \Rightarrow \big(\sigma(s_1), \sigma(s_2)\big) \in R\,]. \tag{13}$$

Группа симметрии — группа перестановок G, которая одновременно является и группой инвариантности и группой автоморфизмов.

Орбиты. *Орбита* [5, 8, 9] состояния $s \in S$ — множество состояний, полученное в результате всех возможных преобразований из Σ для s:

$$\theta(s) = \{\sigma(s) \mid \sigma \in \Sigma\}. \tag{14}$$

Множество всех орбит $O = \{\theta(s) \mid s \in S\}$.

Фактормодель. Для системы переходов $TS = \langle S, S_0, R, P, L \rangle$ с помощью группы симметрии G определим *фактормодель* (англ. quotient model) [5, 9] $TS_G = \langle S_G, S_0^G, R_G, P, L_G \rangle$, где множество состояний $S_G = O$, начальные состояния $S_0^G = \{\theta(s_0) \mid s_0 \in S_0\}$, отношение переходов имеет вид $R_G = \{\big(\theta(s_1), \theta(s_2)\big) \mid (s_1, s_2) \in R\}$, функция разметки $L_G(o) = L(s)$, $o = \theta(s)$.

На рис. 1 представлен пример перехода фактормодели между орбитами: переходы $(s_1, s_2) \in R$ и $\big(\sigma(s_1), \sigma(s_2)\big) = (s_1', s_2') \in R$ модели TS объединяются в один переход $\big(\theta(s_1), \theta(s_2)\big) = (o_1, o_2) \in R_G$ фактормодели TS_G, которая в результате такого объединения в общем случае имеет меньше состояний и переходов, чем исходная модель TS, т. е. $|S_G| \leq |S|$ и $|R_G| \leq |R|$.

Каждое преобразование $\sigma \in \Sigma$ определяет изоморфный граф с вершинами в виде состояний, а вся группа $G = \langle \Sigma, \circ \rangle$ — множество изоморфных графов, которые компактно упакованы в фактормодели. В итоге проблема обнаружения симметрии (выявления орбит) не менее сложна, чем проблема *изоморфизма* графов [8—10].

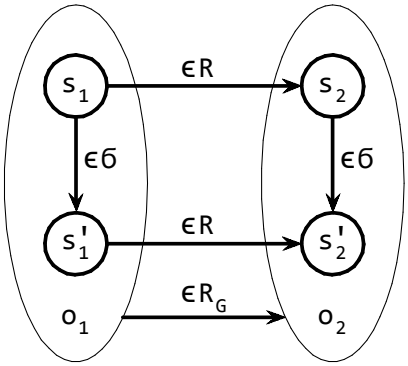

Fig. 1. Transition of the quotient model between orbits

Рис. 1. Переход фактормодели между орбитами

Для системы переходов TS и её фактормодели TS_G, порождённой группой G, существует теорема, аналогичная теореме (10), которая утверждает, что если G является группой симметрии, то для любой формулы φ логики CTL* верно [5, 9]:

$$(TS \models \varphi) \Leftrightarrow (TS_G \models \varphi), \tag{15}$$

т. е. верификация свойства φ может проводиться на фактормодели TS_G. Известно [5, 9], что $TS \equiv TS_G$.

3. Постановка задачи разработки арбитра ресурсов

За основу был взят пример из работы [31], где программное обеспечение марсохода состоит из 11 процессов, которые выполняют различные задачи: вождение марсохода, управление манипулятором робота, создание изображений, связь с Землёй и прочее. Данные процессы во время работы используют 15 общих ресурсов, доступ к которым координирует арбитр. Процессы имеют различные приоритеты доступа к ресурсам. Задача арбитра — обеспечить взаимоисключающий доступ к каждому ресурсу, учитывая приоритеты процессов. Например, связь с Землёй важнее перемещения марсохода, поэтому процесс, обеспечивающий связь, имеет более высокий приоритет, чем процесс управления вождением. Связь должна осуществляться при неподвижном марсоходе, поэтому процесс, обеспечивающий связь, запрашивает сразу два ресурса — приводные двигатели и антенну. Если во время движения марсохода потребуется установить сеанс связи, то арбитр отменит разрешение на использование приводных двигателей для процесса управления вождением и выдаст разрешение на использование двигателей и антенн для процесса, обеспечивающего связь с Землёй.

На рис. 2 показан пример сценария взаимодействия процессов P1 и P2 с арбитром ARB. Процесс P1 делает запрос (Request) ресурса у арбитра ARB, который, в свою очередь, выдаёт разрешение (Grant) на его использование. С этого момента процесс P1 владеет данным ресурсом (период владения выделен серым прямоугольником). Далее более высокоприоритетный процесс P2 делает запрос (Request) этого же ресурса. Арбитр отменяет (Rescind) для процесса P1 разрешение на использование ресурса, которым тот владеет в настоящий момент. Процесс P1 освобождает (Free) ресурс и больше им не владеет. Только после освобождения ресурса процессом P1 арбитр ARB выдаёт разрешение (Grant) на использование этого ресурса более приоритетному процессу P2. С этого момента он является владельцем ресурса. Теперь запрос (Request) ресурса низкоприоритетным процессом P1 отклоняется (Deny) арбитром. Также отклоняется запрос любого процесса, приоритет которого не превышает приоритет процесса P2. При завершении работы с ресурсом процесс P2 освобождает (Free) его и больше им не владеет.

В работе [31] говорится, что модуль арбитра содержит около 3000 строк исходного кода на языке C. Полномасштабная проверка модуля, координирующего работу 11 процессов, конкурирующих

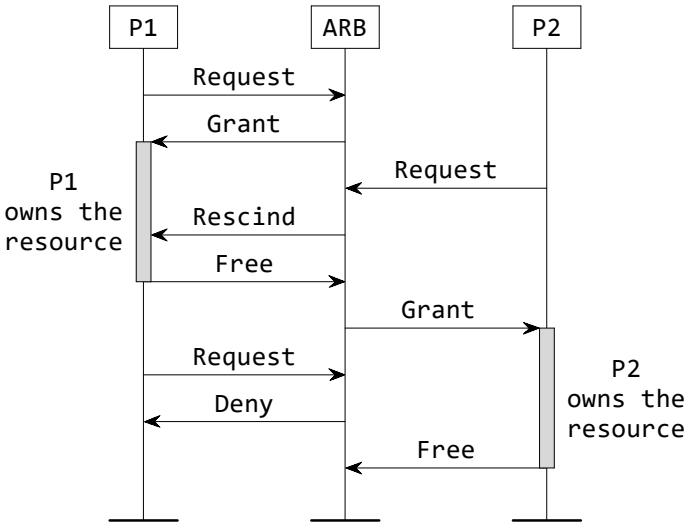

Fig. 2. An example of a scenario for the interaction of processes P1 and P2 with an arbitrator ARB

Рис. 2. Пример сценария взаимодействия процессов P1 и P2 с арбитром ARB

за 15 ресурсов, становится невозможной из-за комбинаторного взрыва в пространстве состояний. Для проверки модуля арбитра используются различные приёмы и абстракции.

Формулировка задачи будет иметь следующий вид. Требуется разработать арбитр ресурсов марсохода для $n \in \mathbb{N}$ процессов и $m \in \mathbb{N}$ ресурсов, где $\mathbb{N}$ — натуральные числа. Процессы имеют два уровня приоритетов: 1 — низкий, 2 — высокий. Все ресурсы независимы — при доступе к ресурсу не требуется запрашивать другие, связанные с ним ресурсы. Главное требование к арбитру — обеспечить взаимоисключающий доступ процессов к ресурсам, т. е. ни один ресурс не должен одновременно использоваться более чем одним процессом.

4. Разработка арбитра ресурсов

Процессы могут запрашивать и освобождать ресурсы, посылая сообщения арбитру. Обозначим $Mes = \{REQ, FREE\}$ множество видов сообщений арбитру от процессов, где REQ и $FREE$ — запрос и освобождение ресурса соответственно. Переменная mes принимает значения из Mes.

Обозначим $Num = \{1, \ldots, n\}$ номера процессов, а $Res = \{1, \ldots, m\}$ номера ресурсов. В каждый конкретный момент времени арбитр будет работать с процессами и ресурсами, которые имеют фиксированные номера из множеств Num и Res соответственно. Однако арбитр будет иметь общий шаблон для всех сценариев поведения независимо от конкретных номеров процессов и ресурсов.

Для примера рассмотрим следующий шаблон поведения: арбитр запретил использование ресурса низкоприоритетному процессу с номером $n_1 \in Num$ и выдал разрешение высокоприоритетному процессу с номером $n_2 \in Num$. Конкретные значения n_1 и n_2 не важны при принятии решения арбитром, главное, чтобы n_1 был номером любого низкоприоритетного процесса, а n_2 — номером любого высокоприоритетного процесса. Таким образом, предполагается наличие некоторой симметрии в поведении арбитра.

Разработаем ядро, которое задаёт поведение арбитра одного ресурса с учётом этой симметрии. Работа с параметрами в виде номеров процессов будет осуществляться вне ядра (в обвязке ядра).

Чтобы отличать процессы друг от друга на уровне ядра будем использовать индексы процессов $I = \{«i», «j», «k»\}$, где «i» — любой процесс, который обращается/обращался к арбитру при свободном ресурсе, «j» — любой процесс (кроме процесса «i»), который обращается/обращался к арбитру при занятом ресурсе процессом «i», «k» — любой процесс (кроме процессов «i» и «j»), который обращается/обращался к арбитру после запроса ресурса более приоритетным процессом «j».

По условию задачи процессы могут иметь всего два приоритета. Поэтому арбитр может отменять разрешение использования ресурса только низкоприоритетным процессам, высокоприоритетным процессам разрешение не отменяется, так как не существует более приоритетных процессов. Таким образом, введённых индексов из I достаточно для построения ядра.

Поведение ядра опишем с помощью конечного автомата Мили, который преобразует информацию, представленную в индексной форме. Назовём его *индексным* автоматом. Формально индексный автомат $A = \langle X, iOut, Q, q_0, \delta, \lambda \rangle$, где множество значений входных сигналов $X = iMes \times \mathbb{B}$, множество индексных сообщений автомату от процессов $iMes = \{req_i, req_j, req_k, free_i, free_j, free_k\}$, где req_i, req_j, req_k — запрос ресурса процессом «i», «j», «k» соответственно, $free_i$, $free_j$, $free_k$ — освобождение ресурса (или отказ от ресурса) процессом «i», «j», «k» соответственно, $\mathbb{B} = \{0, 1\}$ — логические значения. Обозначим *структурный вход* [32] автомата вектором переменных ($imes$, $highPrior$), где $imes$ — сообщение арбитру в индексной форме, $highPrior$ — процесс «j» имеет более высокий приоритет, чем процесс «i». Здесь $imes$ принимает значения из $iMes$, $highPrior$ — из $\mathbb{B}$.

Множество значений выходных сигналов (множество индексных сообщений процессам от автомата) $iOut = \{grant_i, grant_j, deny_i, deny_j, deny_k, rescind_i, \epsilon\}$, где $grant_i$, $grant_j$ — разрешение использования ресурса процессу «i» и «j» соответственно, $deny_i$, $deny_j$, $deny_k$ — отклонение запроса процесса «i», «j», «k» соответственно, $rescind_i$ — отмена разрешения процессу «i» использовать ресурс (требование освободить ресурс), ϵ — отсутствие ответа. Обозначим выход автомата переменной $iout$, принимающей значения из $iOut$.

Множество состояний $Q = \{freed, busy_i, busy_j, wait_j, cancel_j\}$, где $freed$ — ресурс свободен, $busy_i$, $busy_j$ — ресурс занят процессом «i» и «j» соответственно, $wait_j$ — процесс «j» ожидает освобождения ресурса, $cancel_j$ — отказ от ресурса процессом «j». Начальное состояние $q_0 = freed \in Q$. Обозначим текущее состояние автомата переменной q, принимающей значения из Q.

Функции переходов $\delta\colon Q \times X \to Q$ и выходов $\lambda\colon Q \times X \to iOut$ заданы с помощью графа переходов (рис. 3). Изначально автомат находится в состоянии $freed$, что соответствует свободному ресурсу. При этом первый обратившийся процесс будет иметь индекс «i». Автомат при получении запроса ресурса процессом «i» (req_i) выдаёт разрешение на использование этого ресурса ($grant_i$), а также переходит в состояние занятости ресурса процессом «i» ($busy_i$). При освобождении ресурса процессом «i» ($free_i$) автомат переходит в начальное состояние $freed$, выход $iout = \epsilon$.

В состоянии $busy_i$ любой вновь обратившийся к арбитру процесс будет иметь индекс «j». Запрос ресурса req_j низкоприоритетным процессом ($\neg highPrior$) всегда отклоняется ($deny_j$) и автомат остаётся в прежнем состоянии. Если же процесс «j» имеет более высокий приоритет, чем процесс «i» ($highPrior$), то автомат отменяет разрешение процессу «i» использовать ресурс ($rescind_i$) и переходит в состояние $wait_j$ — процесс «j» ожидает освобождения ресурса. Когда процесс «i» освободит ресурс ($free_i$), то автомат выдаёт разрешение использовать данный ресурс процессу «j» ($grant_j$) и переходит в состояние $busy_j$ — ресурс занят процессом «j». В данном состоянии запрос ресурса низкоприоритетным процессом «i» (req_i) отклоняется ($deny_i$). При освобождении ресурса процессом «j» ($free_j$) автомат переходит в начальное состояние $freed$ — ресурс свободен, выход $iout = \epsilon$.

Если процесс «j», ожидая освобождения ресурса (состояние $wait_j$), отказывается от него ($free_j$), то автомат переходит в состояние $cancel_j$ — отказ от ресурса процессом «j». Повторный запрос ресурса (req_j) переводит автомат обратно в состояние $wait_j$. В состоянии $cancel_j$ освобождение ресурса процессом «i» ($free_i$) переводит автомат в начальное состояние $freed$.

В состояниях $wait_j$, $cancel_j$ и $busy_j$ любой вновь обратившийся к арбитру процесс будет иметь индекс «k». Запрос ресурса процессом «k» будет отклонён ($deny_k$) в этих состояниях.

Если на графе переходов в состоянии автомата отсутствует исходящая дуга, соответствующая некоторым значениям входных сигналов, то это значит, что автомат на них не реагирует, т. е. остаётся в том же состоянии, а выход $iout = \epsilon$. Таким образом, автомат A является *вполне определённым* [32],

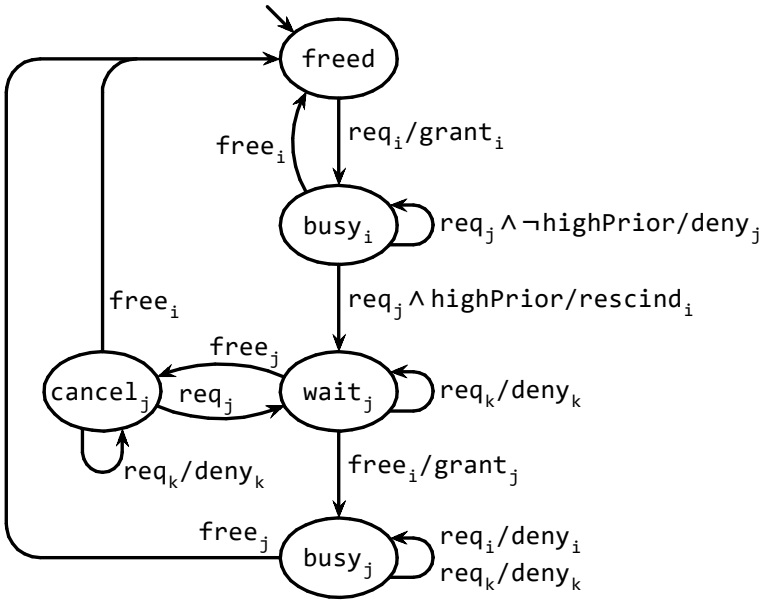

Fig. 3. Index automaton transition graph

Рис. 3. Граф переходов индексного автомата

т. е. для любого состояния и входа определены новое состояние и выход:

$$(\forall q \in Q)(\forall x \in X)(\exists q' \in Q)(\exists iout \in iOut)[q' = \delta(q, x) \wedge iout = \lambda(q, x)].$$

В листинге 1 на языке C++ объявлены используемые в ядре пользовательские типы данных (перечисления): строка 1 содержит объявление сообщений арбитру, строка 2 — возможные индексы процессов, строка 3 — индексные сообщения для автомата, строка 4 — состояния автомата, строка 5 — его выходы. Символу ϵ соответствует значение *empty*.

Листинг 1: Core custom data types. Пользовательские типы данных ядра.

```
1 typedef enum { REQ, FREE } Mes;
2 typedef enum { i, j, k } I;
3 typedef enum { req_i, req_j, req_k, free_i, free_j, free_k } iMes;
4 typedef enum { freed, busy_i, wait_j, busy_j, cancel_j } Q;
5 typedef enum { grant_i, grant_j, deny_i, deny_j, deny_k, rescind_i, empty } iOut;
```

Определим ядро арбитра в виде класса *ArbiterCore* с приватной переменной q для хранения состояния индексного автомата (листинг 2). Для чтения состояния за пределами класса определён публичный метод *getAtmState*.

Листинг 2: Definition of the core class. Определение класса ядра.

```
1 class ArbiterCore {
2   private:
3     Q q = freed;
4   public:
5     Q getAtmState() { return q; }
6     ...
7 };
```

На основе графа (рис. 3) реализуем функции переходов δ и выходов λ автомата A в виде приватного метода *atmIter* класса *ArbiterCore* (листинг 3). Данный метод изменяет состояние q автомата и возвращает его выходные данные.

Листинг 3: atmIter method. Метод atmIter.

```
private:
iOut atmIter(iMes imes, bool highPrior) {
  switch (q) {
    case freed:
      if (imes == req_i) { q = busy_i; return grant_i; } else
      return empty;
    case busy_i:
      if (imes == free_i) { q = freed; return empty; } else
      if (imes == req_j && highPrior) { q = wait_j; return rescind_i; } else
      if (imes == req_j && !highPrior) { return deny_j; } else
      return empty;
    case wait_j:
      if (imes == free_i) { q = busy_j; return grant_j; } else
      if (imes == free_j) { q = cancel_j; return empty; } else
      if (imes == req_k) { return deny_k; } else
      return empty;
    case busy_j:
      if (imes == free_j) { q = freed; return empty; } else
      if (imes == req_i) { return deny_i; } else
      if (imes == req_k) { return deny_k; } else
      return empty;
    case cancel_j:
      if (imes == free_i) { q = freed; return empty; } else
      if (imes == req_j) { q = wait_j; return empty; } else
      if (imes == req_k) { return deny_k; } else
      return empty;
    default: return empty;
  }
}
```

Индексный автомат получает индексное сообщение *imes*, которое принимает значения из множества *iMes*. С помощью функции $indMes\colon I \times Mes \to iMes$ будем выполнять преобразование сообщения *mes* в индексную форму:

$$indMes(ind, mes) = \begin{cases} req_i, \text{ if } mes = REQ \wedge ind = «i»; \\ req_j, \text{ if } mes = REQ \wedge ind = «j»; \\ req_k, \text{ if } mes = REQ \wedge ind = «k»; \\ free_i, \text{ if } mes = FREE \wedge ind = «i»; \\ free_j, \text{ if } mes = FREE \wedge ind = «j»; \\ free_k, \text{ if } mes = FREE \wedge ind = «k». \end{cases}$$

Листинг 4 содержит реализацию данной функции на языке C++.

Листинг 4: C++ code of the indMes function. C++ код функции indMes.

```
private:
iMes indMes(I ind, Mes mes) {
  switch (ind) {
    case i:
      if (mes == REQ)  { return req_i;  }
      if (mes == FREE) { return free_i; }
    case j:
      if (mes == REQ)  { return req_j;  }
      if (mes == FREE) { return free_j; }
    case k:
```

```
11        if (mes == REQ)  { return req_k;  }
12        if (mes == FREE) { return free_k; }
13    }
14  }
```

Структура арбитра представлена на рис. 4. Данная схема отражает состав компонентов арбитра и связи между ними. Видно, что *imes* — результат вычисления функции *indMes*, подаётся на вход индексному автомату (Index Automaton), входящему в состав ядра. Ядро (англ. core) выделено прямоугольником с штриховыми границами.

Индекс *ind* принимает значения из I и вычисляется с помощью функции *getIndex*: $Q \times \mathbb{B}^2 \to I$ (рис. 4), которая имеет вид:

$$getIndex(q, pi, pj) = \begin{cases} «i», \text{ if } (q = freed) \vee pi; \\ «j», \text{ if } (q = busy_i) \wedge \neg pi \vee pj; \\ «k», \text{ else.} \end{cases} \tag{16}$$

Переменная q содержит текущее состояние индексного автомата. Булева переменная $pi = 1$ тогда и только тогда, когда арбитру посылает сообщение i-й процесс. Аналогично определяется значение переменной *pj* для j-го процесса. Таким образом, согласно (16) функция *getIndex* возвращает индекс «i», если арбитру посылает сообщение i-й процесс ($pi = 1$) или ресурс свободен — автомат находится в начальном состоянии ($q = freed$). Функция возвращает индекс «j», если арбитру посылает сообщение j-й процесс ($pj = 1$) или ресурс занят процессом «i» ($q = busy_i$) и сообщение посылает не i-й процесс. Во всех остальных случаях функция возвращает индекс «k».

Согласно схеме на рис. 4 значение переменной *ind* потребуется за пределами ядра (для функции *indUpd*). Поэтому реализуем *getIndex* в виде публичного метода (листинг 5). Данному методу доступно состояние q, поэтому оно не передаётся в качестве аргумента.

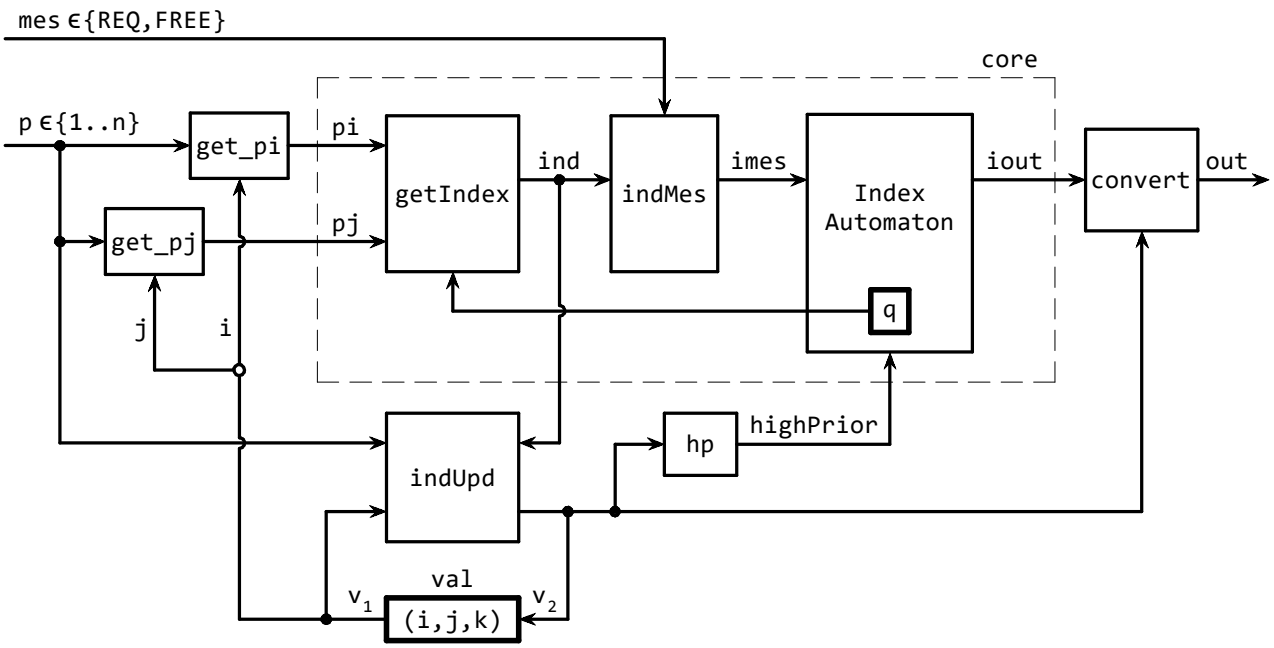

Fig. 4. Arbiter structure

Рис. 4. Структура арбитра

Листинг 5: C++ code of the getIndex method. C++ код метода getIndex.

```
1 public:
2 I getIndex(bool pi, bool pj) {
3   if ((q == freed) || pi) { return i; }
4   else if ((q == busy_i) && !pi || pj) { return j; }
5   else { return k; }
6 }
```

Для вызова ядра определим публичный метод *iter* (листинг 6).

Листинг 6: C++ code of the core iter method. C++ код метода iter ядра.

```
1 public:
2 iOut iter(Mes mes, bool pi, bool pj, bool highPrior) {
3   I ind = getIndex(pi, pj);
4   iMes imes = indMes(ind, mes);
5   iOut iout = atmIter(imes, highPrior);
6   return iout;
7 }
```

На рис. 4 переменные, помещённые в рамки с жирным очертанием (q, i, j, k), должны сохранять свои значения между обработками сообщений, значения остальных переменных вычисляются во время обработки конкретного сообщения на основании имеющихся данных.

Далее переходим к обвязке ядра. Вектор переменных (i, j, k) принимает значения из $V = Num^3$ и сохраняется вне ядра (см. рис. 4). Данный вектор предназначен для хранения номеров процессов, соответствующих одноимённым индексам: переменная i соответствует индексу «i», переменная j — индексу «j», переменная k — индексу «k».

Выход арбитра *out* принимает значения из $Out = (Num \cup \{\epsilon\})^3$, где компоненты вектора содержат номера процессов, получивших определённый ответ от арбитра: первый компонет зарезервирован для выдачи разрешений (*grant*), второй — для отказов (*deny*), третий — для отмены владения ресурсом (*rescind*). Отсутствие ответа обозначается символом ϵ.

Определим вектор переменных (i, j, k) и выход арбитра *out* как пользовательские типы данных (листинг 7). В строке 1 объявлена структура для хранения значений индексов — вектора (i, j, k). Для кодирования отсутствия ответа (символа ϵ) выбрано значение -1 (строка 6). В строке 7 определён выход арбитра, в строке 11 — метод для его сброса.

Листинг 7: Custom data types outside the core. Пользовательские типы данных вне ядра.

```
1  struct V {
2    int i = 0;
3    int j = 0;
4    int k = 0;
5  };
6  const int EMPTY = -1; // No response.
7  struct Out {
8    int grant = EMPTY;
9    int deny = EMPTY;
10   int rescind = EMPTY;
11   void reset() {
12     grant = EMPTY;
13     deny = EMPTY;
14     rescind = EMPTY;
15   }
16 };
```

Определим арбитр одного ресурса в виде класса *OneResArbiter* (листинг 8), который содержит недоступные снаружи (строка 2) ядро (строка 3), индексы (строка 4) и выход (строка 5).

Листинг 8: Defining of the one resource arbiter class. Определение класса арбитра одного ресурса.

```
class OneResArbiter {
  private:
    ArbiterCore core;
    V val; // Index values.
    Out out;
    ...
};
```

Вектор переменных (i, j, k) обновляется с помощью функции $indUpd\colon Num \times I \times V \to V$, которая имеет вид:

$$indUpd\big(p, ind, (i, j, k)\big) = \begin{cases} (p, j, k), \text{ if } ind = «i»; \\ (i, p, k), \text{ if } ind = «j»; \\ (i, j, p), \text{ if } ind = «k». \end{cases} \tag{17}$$

Например, ядро определило, что текущему процессу с номером $p \in Num$ соответствует индекс $ind = «i»$, тогда согласно (17) значение переменной i изменится на значение p, а переменные j и k не меняют своих значений. На рис. 4 вектор v_1 отражает старое значение (i, j, k), а вектор v_2 — вновь вычисленное. В классе *OneResArbiter* реализуем обновление вектора с помощью одноимённого приватного метода (листинг 9), который не принимает и не возвращает значений из V, а работает с вектором *val* напрямую. Аргумент функции p переименован в *proc*.

Листинг 9: C++ code of the indUpd method. C++ код метода indUpd.

```
private:
void indUpd(int proc, I ind) {
  if (ind == i) { val.i = proc; }
  if (ind == j) { val.j = proc; }
  if (ind == k) { val.k = proc; }
}
```

Индексный выход *iout* автомата преобразуется в выход *out* арбитра (см. рис. 4) с помощью функции $convert\colon iOut \times V \to Out$, которая имеет вид:

$$convert\big(iout, (i, j, k)\big) = \begin{cases} (i, \epsilon, \epsilon), \text{ if } (iout = grant_i); \\ (j, \epsilon, \epsilon), \text{ if } (iout = grant_j); \\ (\epsilon, i, \epsilon), \text{ if } (iout = deny_i); \\ (\epsilon, j, \epsilon), \text{ if } (iout = deny_j); \\ (\epsilon, k, \epsilon), \text{ if } (iout = deny_k); \\ (\epsilon, \epsilon, i), \text{ if } (iout = rescind_i); \\ (\epsilon, \epsilon, \epsilon), \text{ if } (iout = \epsilon). \end{cases}$$

В классе *OneResArbiter* реализуем одноимённый приватный метод (листинг 10), который не принимает на вход вектор (i, j, k), а работает с вектором *val* напрямую, также метод не возвращает значение из *Out*, а непосредственно модифицирует вектор *out*.

Листинг 10: C++ code of the convert method. C++ код метода convert.

```
private:
void convert(iOut iout) {
  out.reset();
  if (iout == grant_i) { out.grant = val.i; }
  if (iout == grant_j) { out.grant = val.j; }
  if (iout == deny_i)  { out.deny  = val.i; }
```

```
7   if (iout == deny_j)  { out.deny  = val.j; }
8   if (iout == deny_k)  { out.deny  = val.k; }
9   if (iout == rescind_i) { out.rescind = val.i; }
10 }
```

Индексный автомат получает на вход значение булевой переменной *highPrior* (см. рис. 4), которая указывает, что процесс «j» имеет более высокий приоритет, чем процесс «i». Её значение вычисляется с помощью функции $hp\colon V \to \mathbb{B}$, которая имеет вид:

$$hp\big((i,j,k)\big) = \big(prior(j) > prior(i)\big),$$

где функция $prior\colon Num \to \{1,2\}$ возвращает приоритет процесса. Реализуем функцию *hp* в виде одноимённого приватного метода (листинг 11, строка 2), кторый не принимает в качестве аргумента вектор (i,j,k), а работает с вектором *val* напрямую. Приоритеты процессов будем хранить в массиве (строка 1). Для примера инициализирован массив из 10 элементов.

Листинг 11: C++ code of the hp method. C++ код метода hp.

```
1 const int prior[10] = { 1, 1, 2, 1, 1, 1, 1, 1, 1, 2 };
2 private:
3 bool hp() { return (prior[val.j] > prior[val.i]); }
```

Значения переменных *pi* и *pj* (см. рис. 4) определяются с помощью функций, отображающих множество Num^2 в $\mathbb{B}$. Функция $get\_pi(p,i) \equiv (p = i)$, функция $get\_pj(p,j) \equiv (p = j)$. Здесь p — номер пославшего сообщение процесса, i и j — номера процессов, которые соответствуют процессам «i» и «j» соответственно. Реализуем данные функции в виде одноимённых методов (листинг 12), которые используют вектор *val* напрямую. Аргумент функции p переименован в *proc*.

Листинг 12: C++ code of methods for calculating variables pi and pj. C++ код методов для вычисления переменных pi и pj.

```
1 private:
2 bool get_pi(int proc) { return (proc == val.i); }
3 bool get_pj(int proc) { return (proc == val.j); }
```

Итерация вычислений арбитра ресурса определена в приватном методе *iter* (листинг 13). Изначально вычисляются значения переменных *pi* и *pj* (строки 3 и 4), далее на их основании — индекс *ind* (строка 5). С помощью метода *indUpd* обновляется значение переменной *val* (строка 6). Вычисляется значение переменной *highPrior* (строка 7). Вызов ядра фиксирует индексный выход *iout* (строка 8), который конвертируется в выход арбитра (строка 9).

Листинг 13: C++ code of the resource arbiter iter method. C++ код метода iter арбитра ресурса.

```
1  private:
2  void iter(Mes mes, int proc) {
3    bool pi = get_pi(proc);
4    bool pj = get_pj(proc);
5    I ind = core.getIndex(pi, pj);
6    indUpd(proc, ind);
7    bool highPrior = hp();
8    iOut iout = core.iter(mes, pi, pj, highPrior);
9    convert(iout);
10 }
```

Для обращения к арбитру ресурса извне определим ряд публичных методов (листинг 14). Метод *req* (строка 4) предназначен для запроса ресурса. В качестве параметра ему передаётся номер процесса *proc*, который запрашивает ресурс. Если номер процесса находится в допустимом диапазоне, то вызывается метод *iter* с соответствующими параметрами. Приватный метод *inRange* проверяет

допустимость номера процесса: минимальное значение равно 0 (в программе процессы нумеруем не с 1, а с 0), максимальное значение ограничено размером массива, содержащего приоритеты процессов. Для освобождения ресурса вызывается метод *free* (строка 5), который работает аналогично методу *req*.

Листинг 14: C++ code for public methods of the ResArbiter class. C++ код публичных методов класса ResArbiter.

```
1  private:
2    bool inRange(int proc) { return (proc >= 0) && (proc < sizeof(prior)); }
3  public:
4    void req(int proc)  { if (inRange(proc)) iter(REQ, proc); }
5    void free(int proc) { if (inRange(proc)) iter(FREE, proc); }
6    int getGrantProc()   { return out.grant; }
7    int getDenyProc()    { return out.deny; }
8    int getRescindProc() { return out.rescind; }
9    int owner() {
10     Q q = core.getAtmState();
11     switch (q) {
12       case Q::busy_i:
13       case Q::wait_j:
14       case Q::cancel_j: return val.i;
15       case Q::busy_j: return val.j;
16       case Q::freed:
17       default: return EMPTY;
18     }
19   }
20   int waiting() {
21     Q q = core.getAtmState();
22     if (q == Q::wait_j) return val.j;
23     else return EMPTY;
24   }
```

Методы *getGrantProc* (строка 6), *getDenyProc* (строка 7), *getRescindProc* (строка 8) возвращают номера процессов, получивших разрешение, отказ и отмену использования ресурса соответственно. Метод *owner* (строка 9) возвращает номер процесса, владеющего ресурсом. Если индексный автомат находится в состояниях $busy_i$, $wait_j$ или $cancel_j$, то владельцем является «i»-й процесс (строки 12-14). В состоянии $busy_j$ владельцем является «j»-й процесс (строка 15). В остальных случаях у ресурса нет владельца, поэтому возвращается отсутствие ответа ϵ (строки 16, 17).

Метод *waiting* (строка 20) возвращает номер процесса, ожидающего освобождения ресурса. Если индексный автомат находится в состоянии $wait_j$, то освобождения ресурса ожидает «j»-й процесс (строка 22). В остальных случаях ожидающих процессов нет, поэтому возвращается отсутствие ответа ϵ (строка 23).

Далее в виде класса *Arbiter* определим арбитр для множества ресурсов (листинг 15). Константа *NUMRES* задаёт количество ресурсов (строка 1). Класс содержит массив *arb* арбитров одного ресурса *OneResArbiter*, содержащий равное *NUMRES* количество элементов (строка 4). Во всех публичных методах класса (*req* — строка 7, *free* — строка 10, *getGrantProc* — строка 13, *getDenyProc* — строка 17, *getRescindProc* — строка 21, *owner* — строка 25, *waiting* — строка 29), если номер ресурса находится в допустимом диапазоне, то вызывается одноимённый метод арбитра соответствующего ресурса. Проверка допустимости номеров ресурсов выполняется с помощью вспомогательной функции *inRange* (строка 5). Полный исходный код представленного решения находится в общем доступе: https://github.com/MaximNeyzov/rover-resources-arbiter.

Листинг 15: Defining of the resources arbiter class. Определение класса арбитра ресурсов.

```
const int NUMRES = 10; // Number of resources.
class Arbiter {
  private:
    OneResArbiter arb[NUMRES];
    bool inRange(int res) { return ((res >= 0) && (res < NUMRES)); }
  public:
    void req(int proc, int res) {
      if (inRange(res)) arb[res].req(proc);
    }
    void free(int proc, int res) {
      if (inRange(res)) arb[res].free(proc);
    }
    int getGrantProc(int res) {
      if (inRange(res)) return arb[res].getGrantProc();
      else return EMPTY;
    }
    int getDenyProc(int res) {
      if (inRange(res)) return arb[res].getDenyProc();
      else return EMPTY;
    }
    int getRescindProc(int res) {
      if (inRange(res)) return arb[res].getRescindProc();
      else return EMPTY;
    }
    int owner(int res) {
      if (inRange(res)) return arb[res].owner();
      else return EMPTY;
    }
    int waiting(int res) {
      if (inRange(res)) return arb[res].waiting();
      else return EMPTY;
    }
};
```

Технически арбитр может координировать доступ к ресурсам среди процессов или потоков операционной системы. Независимо от выбранного варианта требуется подходящим образом обеспечить атомарность выполнения всех публичных методов класса *Arbiter*.

В рассмотренном примере ядро арбитра содержит один индексный автомат. При разработке более сложных программ ядро может быть представлено, например, как композиция автоматов.

5. Анализ моделей поведения арбитра и ядра

5.1. Модель поведения арбитра

Модель поведения арбитра зададим в виде системы переходов $TS = \langle S, S_0, R, P, L\rangle$, где множество состояний $S \subseteq Mes \times Num \times I \times \mathbb{B} \times Q \times V \times iOut \times Out$. Обозначим состояние $s \in S$ в виде вектора $(mes, p, ind, highPrior, q, v, iout, out) \in S$. Здесь $mes \in Mes$ — сообщение арбитру от процесса (отражает вызванный в программе метод: *req* или *free*). Значение аргумента *proc* данных методов представляет собой $p \in Num$ — номер процесса. Значение переменной *val* представляет компонент $v \in V$. Остальные компоненты вектора представляют собой значения одноимённых переменных C++ программы: $ind \in I$ — индекс процесса, $highPrior \in \mathbb{B}$ — наличие высокоприоритетного процесса, $q \in Q$ — состояние индексного автомата, $iout \in iOut$ — значение выхода индексного автомата, $out \in Out$ — значение выхода арбитра.

Определим одно начальное состояние: $S_0 = \{s_0\}$, где $s_0 = (free, 1, «i», 0, q_0, (1, 1, 1), \epsilon, (\epsilon, \epsilon, \epsilon))$.

Введём обозначение $pr_i(w)$ — проекция вектора w на i-ю ось, тогда согласно структуре арбитра на рис. 4 отношение переходов имеет вид:

$$\begin{aligned} R = \big\{(s_1, s_2) \in S \times S \,|\, & [mes \in Mes] \wedge [p \in Num] \wedge \\ & [pi = get\_pi(p, i)] \wedge [i = pr_1(v_1)] \wedge [v_1 = pr_6(s_1)] \wedge \\ & [pj = get\_pj(p, j)] \wedge [j = pr_2(v_1)] \wedge \\ & [ind = getIndex(q_1, pi, pj)] \wedge [q_1 = pr_5(s_1)] \wedge \\ & [imes = indMes(ind, mes)] \wedge \\ & [v_2 = indUpd(p, ind, v_1)] \wedge [highPrior = hp(v_2)] \wedge \\ & [q_2 = \delta(q_1, x)] \wedge [x = (imes, highPrior)] \wedge [iout = \lambda(q_1, x)] \wedge \\ & [out = convert(iout, v_2)] \wedge \\ & [s_2 = (mes, p, ind, highPrior, q_2, v_2, iout, out)]\big\}. \end{aligned} \tag{18}$$

Множество достижимых состояний $S = \{s \,|\, (\exists \pi \in \Pi_{TS})\ (\exists i \in \mathbb{N}_0)\ \pi(i) = s\}$. Множество атомарных утверждений $P = Mes \cup I \cup \{highPrior\} \cup Q \cup iOut \setminus \{\epsilon\}$ предназначено для оценки состояния ядра.

Функция разметки состояний атомарными утверждениями $L\colon S \to 2^P$ имеет вид: $L(s) = L_1(s) \cup L_2(s) \cup L_3(s) \cup L_4(s) \cup L_5(s)$, где $L_1(s) = \{pr_1(s)\}$, $L_2(s) = \{pr_3(s)\}$, $L_4(s) = \{pr_5(s)\}$,

$$L_3(s) = \begin{cases} \{highPrior\}, & \text{if } pr_4(s) = 1; \\ \varnothing, & \text{else.} \end{cases}$$

$$L_5(s) = \begin{cases} \{pr_7(s)\}, & \text{if } pr_7(s) \neq \epsilon; \\ \varnothing, & \text{else.} \end{cases}$$

5.2. Модель поведения ядра

Модель поведения ядра арбитра зададим в виде системы переходов $TS' = \langle S', S'_0, R', P, L' \rangle$, которая является проекцией системы переходов TS, выполненной с помощью *сюръективного* отображения $d\colon S \to S'$ вида:

$$d(mes, p, ind, highPrior, q, v, iout, out) = (mes, ind, highPrior, q, iout), \tag{19}$$

т. е. из проекции удалены (англ. delete) компоненты p, v, out, не описывающие состояние ядра.

Определим одно начальное состояние: $S'_0 = \{s'_0\}$, где

$$s'_0 = d(s_0) = (free, \text{«}i\text{»}, 0, q_0, \epsilon). \tag{20}$$

Отношение переходов R' формируется из отношения R путем преобразования его переходов с помощью d:

$$R' = \big\{\big(d(s_1), d(s_2)\big) \in S' \times S' \,|\, (s_1, s_2) \in R\big\}. \tag{21}$$

Функция разметки состояний атомарными утверждениями $L'\colon S' \to 2^P$ определяется следующим образом:

$$(\forall s \in S)[L'\big(d(s)\big) = L(s)], \tag{22}$$

так как отображение $d\colon S \to S'$ удаляет компоненты вектора $s \in S$, которые не используются при оценке истинности атомарных утверждений из P.

5.3. Связь моделей поведения арбитра и ядра

Установим соответствие $B \subseteq S \times S'$ между множествами состояний систем переходов TS и TS':

$$(\forall s \in S)\, B(s, d(s)). \tag{23}$$

Из (22) и (23) следует условие *совпадения разметки* соответствующих состояний (5). Каждому переходу в системе переходов TS соответствует переход в системе переходов TS', и наоборот. Условия *взаимного соответствия переходов* (6) и (7) следуют из (21). Согласно (5), (6) и (7) B является отношением бисимуляции.

Условия *взаимного соответствия начальных состояний* (8) и (9) следуют из (20) и (23). Таким образом, имеем бисимуляционную эквивалентность: $TS \equiv TS'$.

5.4. Группа симметрии модели поведения арбитра

Поведение арбитра симметрично относительно номеров процессов: изменение нумерации процессов сохраняет темпоральные свойства. Ядро задаёт некоторую группу симметрии G: набор преобразований Σ формируется с помощью биективных функций $F = \{f : Num \to Num\}$ перенумерации процессов. Общее число функций перенумерации $|F| = |Num|!$, где «!» — факториал числа.

Функция $\sigma \in \Sigma$ формируется на основе функции $f \in F$, где $|\Sigma| = |F|$. Вектор $v = (i, j, k) = pr_6(s)$ заменяется на $v' = \big(f(i), f(j), f(k)\big)$, значение p — на $p' = f(p)$. Также компонент out заменяется на $out' = convert(iout, v')$. Остальные компоненты вектора состояния $s \in S$ не изменяются. Получим

$$\sigma\big((mes, p, ind, highPrior, q, v, iout, out)\big) = (mes, p', ind, highPrior, q, v', iout, out').$$

Приоритеты процессов не изменяются после их перенумерации и определяются с помощью функции $prior'(p') = prior(p)$, где p и p' — номера процессов до и после перенумерации соответственно, т. е. $p' = f(p)$. В итоге компонент $highPrior$ вектора состояния $s \in S$ не изменяется.

Таким образом, преобразование $\sigma \in \Sigma$, построенное с помощью перенумерации процессов $f \in F$, не приводит к изменению состояния ядра $s' = (mes, ind, highPrior, q, iout) \in S'$, поэтому верно:

$$(\forall \sigma \in \Sigma)\, (\forall s \in S)\, \big[d(s) = d\big(\sigma(s)\big)\big]. \tag{24}$$

Семейство переходов. Согласно (21) каждому переходу из R' соответствует семейство переходов из R, которое определим с помощью функции $Fam\colon R' \to 2^R$ следующего вида:

$$Fam\big((s'_1, s'_2)\big) = \big\{(s_1, s_2) \in R \,|\, s'_1 = d(s_1) \wedge s'_2 = d(s_2)\big\}. \tag{25}$$

Перенумерация процессов изменяет переход внутри семейства:

$$(\forall \sigma \in \Sigma)\, \big(\forall (s_1, s_2) \in R\big)\, \big(\forall (s'_1, s'_2) \in R'\big)\, \big[(s_1, s_2) \in Fam\big((s'_1, s'_2)\big) \Rightarrow \big(\sigma(s_1), \sigma(s_2)\big) \in Fam\big((s'_1, s'_2)\big)\big]. \tag{26}$$

Утверждение (26) следует из (24) и (25). Продемонстрируем истинность формулы (26) на примере (рис. 5), который иллюстрирует связи между арбитром, ядром и автоматом. Фрагмент индексного автомата A (изображён справа) содержит два перехода: первый — между состояниями *freed* и $busy_i$, второй — между $busy_i$ и $wait_j$. Каждому переходу автомата A соответствует переход ядра TS' (соответствие состояний автомата и ядра указано горизонтальными штриховыми линиями). Например, начальному состоянию $q_0 \in Q$ автомата A соответствует состояние $s'_0 \in S'$ ядра TS'. Графическое обозначение состояния ядра (TS' state notation) представлено над фрагментом системы переходов TS'. В первой строке указано сообщение $mes \in Mes$ и индекс процесса $ind \in I$, во второй — индексный выход $iout \in iOut$, в третьей — состояние автомата $q \in Q$. Например, в начальном состоянии $s'_0 \in S'$ ядра TS' $mes = free$, $ind = «i»$, $iout = \epsilon$, $q = freed$.

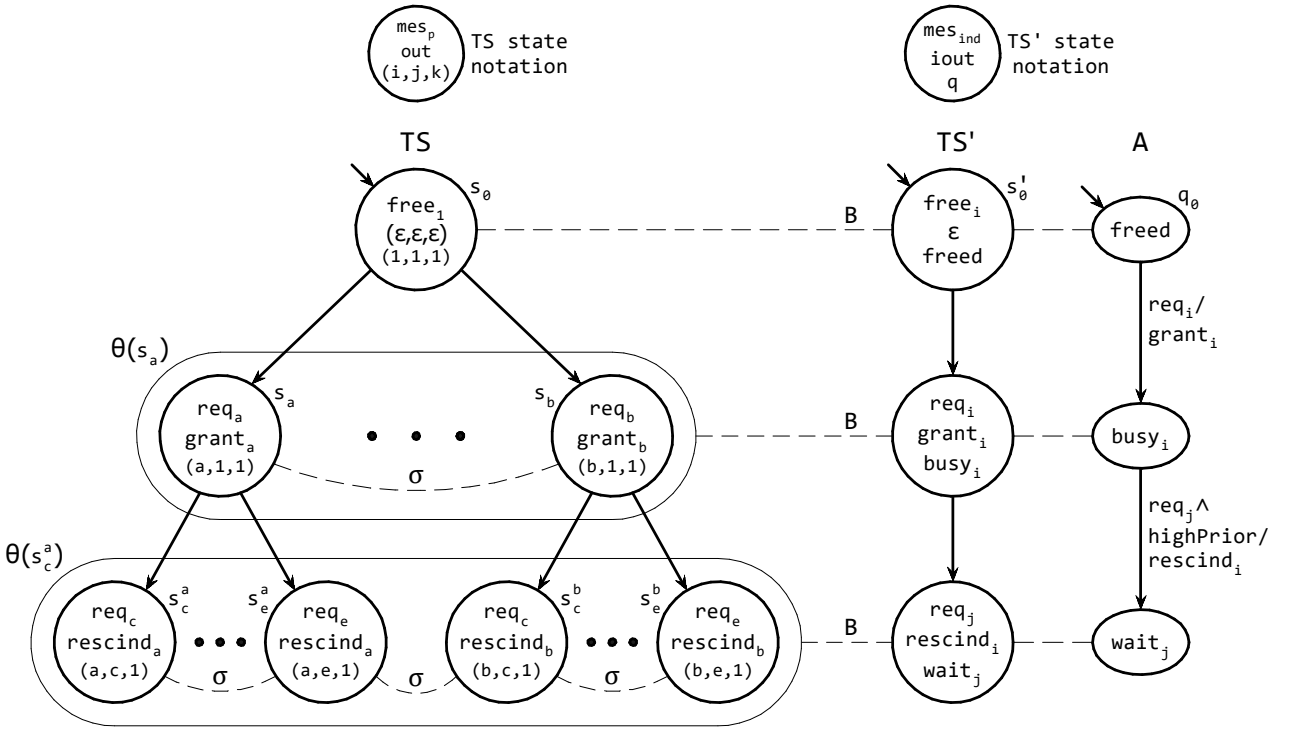

Fig. 5. Illustration of the relationship between the arbiter, the core and the automaton

Рис. 5. Иллюстрация взаимосвязи между арбитром, ядром и автоматом

Каждому переходу ядра TS' соответствует семейство переходов арбитра TS (соответствие B состояний ядра и выделенных овалом семейств состояний арбитра указаны горизонтальными штриховыми линиями). Например, начальному состоянию $s'_0 \in S$ ядра TS' соответствует состояние $s_0 \in S$ арбитра TS. Графическое обозначение состояния арбитра (TS state notation) представлено над фрагментом системы переходов TS. В первой строке указано сообщение $mes \in Mes$ и номер процесса $p \in Num$, во второй — выход $out \in Out$ арбитра, в третьей — вектор номеров процессов $v = (i, j, k) \in V = Num^3$. Например, в начальном состоянии $s_0 \in S$ арбитра TS $mes = free$, $p = 1$, $out = (\epsilon, \epsilon, \epsilon)$, $(i, j, k) = (1, 1, 1)$.

Пусть переходу автомата A из состояния $busy_i$ в состояние $wait_j$ соответствует переход арбитра TS из состояния s_a в состояние s^a_c, в котором $i = a$, а $j = c$ (см. рис. 5). Допустим, что функция перенумерации процессов $f(a) = b$, а $f(c) = e$, тогда $\sigma(s_a) = s_b$, а $\sigma(s^a_c) = s^b_e$, в котором $i = b$, а $j = e$. Таким образом, каждое состояние, полученное в результате преобразования $\sigma \in \Sigma$, принадлежат своему семейству состояний, т. е. состояния внутри семейства получаются в результате преобразований $\sigma \in \Sigma$ — на рис. 5 это изображено в виде изогнутых штриховых линий. В результате рассмотренной перенумерации процессов $f \in F$ переход $r_1 = (s_a, s^a_c) \in R$ был заменён на переход $r_2 = (s_b, s^b_e) \in R$. Так как выбранная нумерация процессов не оказывает влияния на поведение автомата A и ядра TS', то переход $r_2 \in R$ так же, как и переход $r_1 \in R$, порождается переходом автомата A из состояния $busy_i$ в состояние $wait_j$.

Группа G, построенная на основе функций перенумерации F, сохраняет разметку состояний, т. е является группой инвариантности (11), так как атомарные утверждения $p \in P$ отражают состояние ядра, которое не изменяется при преобразовании $\sigma \in \Sigma$. Утверждение (11) следует из (24). Также группа G сохраняет отношение переходов R (18), т. е является группой автоморфизмов (13). Утверждение (13) следует из (26). В итоге получили, что G — группа симметрии.

Рассмотренные ранее семейства состояний (выделены овалами на рис. 5) представляют собой орбиты: орбита $\theta(s_a)$ состояния s_a и орбита $\theta(s^a_c)$ состояния s^a_c.

Каждое состояние орбиты $o \in O$ находится в отношении B с одним единственным состоянием из S':

$$(\forall o \in O)\ (\forall s_1, s_2 \in o)\ (\forall s'_1, s'_2 \in S')\ [B(s_1, s'_1) \wedge B(s_2, s'_2) \Rightarrow s'_1 = s'_2], \tag{27}$$

т. е. каждой орбите $o \in O$ соответствует одно состояние из S' (см. рис. 5). Утверждение (27) следует из (23) и (24). Таким образом, фактормодель TS_G может быть заменена на модель поведения ядра TS'. Все три модели бисимуляционно эквивалентны: $TS \equiv TS' \equiv TS_G$.

6. Верификация ядра арбитра

Бисимуляционная эквивалентность моделей поведения арбитра и ядра ($TS \equiv TS'$) сохраняется при условии, что TS' является проекцией TS. Таким образом, при верификации ядра TS' требуется обеспечить такое же поведение на его входах, как если бы данное ядро работало в составе арбитра. Однако такое точное воспроизведение поведения входов ядра может вызвать затруднения, поэтому будем использовать абстракцию данного поведения: вектор входов ядра (*mes*, *pi*, *pj*, *highPrior*) (см. рис. 5) будет иметь *абсолютно недетерминированное* [33] поведение, т. е. в любой момент времени на вход ядра может поступать любой элемент из множества $Mes \times \mathbb{B}^3$. Система переходов TS'', полученная в результате данной абстракции, является *симуляцией* системы переходов TS', т. е. $TS' \preceq TS''$.

Для проверки свойств применяется инструмент проверки модели Spin[1], который позволяет выполнять автоматическое извлечение модели из C-кода [31, 34]. Воспользуемся данной возможностью — преобразуем C++ код ядра в C-код. Для этого скопируем код, находящийся внутри класса *ArbiterCore*, удалим из него все модификаторы доступа *private* и *public*, а также метод *getAtmState*. Теперь есть прямой доступ к переменной состояния автомата q и всем функциям — бывшим методам класса. Результирующий код сохраним в файле «*Core.c*». Так как язык C не поддерживает булев тип данных, то добавим его в качестве пользовательского типа данных ядра (листинг 16).

Листинг 16: Boolean data type in C code. Булев тип данных в C-коде.

```
1  typedef enum { false = 0, true = 1 } bool;
```

Promela-код модели ядра для Spin представлен в листинге 17. В строке 1 с помощью C-декларации (ключевое слово *c_decl*) подключен файл «*Core.c*», содержащий C-код ядра. Вектор входов ядра (*mes*, *pi*, *pj*, *highPrior*) задекларирован в строках 3-8, вектор выходов (*ind*, *iout*) — в строках 10-13. Состояние автомата q здесь не декларируется, так как уже содержится в файле «*Core.c*».

Согласно (19) вектор состояния модели имеет вид (*mes*, *ind*, *highPrior*, *q*, *iout*). Объявим набор данных переменных в качестве компонент вектора состояния модели (строки 15-19). Ключевое слово *c_track* указывает, что переменная должна *отслеживаться* при верификации (её значение в процессе поиска в глубину должно восстанавливаться при возврате к более раннему состоянию [31]). С помощью символа «&» указывается адрес переменной. Далее следует размер памяти в байтах. Слово «*Matched*» указывает, что при сравнении двух состояний должны *сопоставляться* значения данной переменной, при «*UnMatched*» значения переменной будут игнорироваться [31]. Таким образом, все компоненты вектора состояния модели будут отслеживаться и сопоставляться при верификации. Входные переменные *pi* и *pj* не сопоставляются и не отслеживаются, так как не входят в вектор состояния модели.

Листинг 17: Promela code of the core model. Promela-код модели ядра.

```
1  c_decl{ #include "Core.c" };
2  // Core inputs:
3  c_decl{
4    Mes mes = FREE;
5    bool pi = false;
```

[1] https://spinroot.com/

```
6     bool pj = false;
7     bool highPrior = false;
8   };
9   // Core outputs:
10  c_decl{
11    I ind = i;
12    iOut iout = empty;
13  };
14  // Components of the model state vector:
15  c_track "&mes" "sizeof(char)" "Matched";
16  c_track "&ind" "sizeof(char)" "Matched";
17  c_track "&highPrior" "sizeof(char)" "Matched";
18  c_track "&q" "sizeof(char)" "Matched"; // "q" defined in "Core.c"
19  c_track "&iout" "sizeof(char)" "Matched";
20  active proctype model() {
21    do
22    :: atomic {
23        if
24        :: c_code{ mes = REQ;  };
25        :: c_code{ mes = FREE; };
26        fi;
27        if
28        :: c_code{ pi = false; };
29        :: c_code{ pi = true;  };
30        fi;
31        if
32        :: c_code{ pj = false; };
33        :: c_code{ pj = true;  };
34        fi;
35        if
36        :: c_code{ highPrior = false; };
37        :: c_code{ highPrior = true;  };
38        fi;
39        c_code {
40          ind = getIndex(pi, pj);
41          iout = iter(mes, pi, pj, highPrior);
42        };
43       }
44    od
45  }
```

Promela-код задаёт один активный процесс *model* (строка 20), который циклически (цикл *do* – *od*, строки 21-44) выполняет одно атомарное (*atomic*, строка 22) действие, которое заключается в недетерминированном выборе значений входных переменных *mes* (строки 23-26), *pi* (строки 27-30), *pj* (строки 31-34), *highPrior* (строки 35-38) и вычислении выходных переменных *ind* (строка 40) и *iout* (строка 41). Внутренняя переменная *q* может изменить своё значение в результате вызова функции *iter* (строка 41). Все действия по изменению значений переменных представлены в виде C-кода и заключены в секции с помощью ключевого слова *c_code*.

В итоге представленный в листинге 17 Promela-код задаёт недетерминированное поведение входов ядра, а также указывает способ извлечения модели ядра из C-кода.

Свойства для верификации. Главное свойство арбитра — обеспечение взаимоисключающего доступа процессов к ресурсу. Формализуем данное свойство в виде двух LTL-формул:

$$\varphi_1 \stackrel{\text{def}}{=} \mathbf{G}\big(\, grant_i \Rightarrow \mathbf{X}(\neg grants\ \mathbf{W}\ free_i)\,\big),$$
$$\varphi_2 \stackrel{\text{def}}{=} \mathbf{G}\big(\, grant_j \Rightarrow \mathbf{X}(\neg grants\ \mathbf{W}\ free_j)\,\big),$$

где $grants \stackrel{\text{def}}{=} grant_i \vee grant_j$. Свойство φ_1 означает, что всегда, если выдано разрешение использовать ресурс ($grant_i$), то со следующего момента времени разрешения больше не выдаются ($\neg grants$) до тех пор, пока ресурс не будет освобождён ($free_i \stackrel{\text{def}}{=} FREE \wedge «i»$). Освобождать ресурс должен именно тот процесс, который им владеет. Свойство φ_2 означает то же, что и свойство φ_1, только для «j»-го процесса.

Следующее свойство — доступность свободного ресурса. Всегда запрос (REQ) свободного ресурса ($freed$) сопровождается разрешением на его использование ($grant_i$):

$$\varphi_3 \stackrel{\text{def}}{=} \mathbf{G}\big(freed \wedge \mathbf{X}(REQ) \Rightarrow \mathbf{X}(grant_i) \big).$$

Последнее свойство — доступность ресурса для высокоприоритетного процесса:

$$\varphi_4 \stackrel{\text{def}}{=} cond \Rightarrow \mathbf{G}\big(busy_i \wedge \mathbf{X}(req_j \wedge highPrior) \Rightarrow \mathbf{XF}(grant_j) \big),$$

где $cond \stackrel{\text{def}}{=} \mathbf{GF}(free_i) \wedge \mathbf{G}(\neg free_j)$ — условия для выполнения данного свойства, Таким образом, требуется, чтобы всегда в будущем низкоприоритетный («i»-й) процесс освобождал ресурс, а также высокоприоритетный («j»-й) процесс никогда не отказывался от ресурса ($free_j \stackrel{\text{def}}{=} FREE \wedge «j»$). Само свойство утверждает, что всегда если высокоприоритетный ($highPrior$) процесс запрашивает ($req_j \stackrel{\text{def}}{=} REQ \wedge «j»$) занятый ($busy_i$) ресурс, то он неизбежно его получит ($grant_j$) в будущем.

Отметим, что свойства $\varphi_1, \ldots, \varphi_4$ задают поведение в общем виде, т. е. для любых процессов с номерами $i, j \in Num = \{1, \ldots, n\}$. При этом формализация свойств не содержит ни множества Num, ни квантора всеобщности. Данное упрощение обусловлено использованием индексного автомата, который реализует координацию процессов в абстрактном виде, не оперируя фактическими номерами процессов. Формально семантика индексов «i» и «j» определена в C++ коде арбитра: индекс ind зависит от состояния q, номера процесса p, значений i, j и вычисляется с помощью функций $getIndex$, get_pi, get_pj, функция $indUpd$ отвечает за обновление значений i, j (см. рис. 4).

Результаты верификации. Свойства $\varphi_1, \ldots, \varphi_4$ были проверены верификатором Spin за несколько секунд на персональном компьютере с процессором Intel Core i5-3570 3.4 ГГц и 8 ГБ оперативной памяти. Все свойства выполняются: $TS'' \models \varphi_x$, где $x = 1, \ldots, 4$. Имеем $(TS'' \models \varphi_x) \Rightarrow (TS' \models \varphi_x)$ согласно (4) и $(TS' \models \varphi_x) \Rightarrow (TS \models \varphi_x)$ согласно (10). Таким образом, модели поведения ядра (TS') и арбитра (TS) соответствуют предъявляемым к ним требованиям, выраженным в виде LTL-формул $\varphi_1, \ldots, \varphi_4$.

Заключение

В работе представлен подход к разработке и верификации программных реагирующих систем. Суть подхода состоит в том, что основная часть программы, называемая ядром, проектируется с учётом симметрии так, чтобы гарантировать соблюдение заданных темпоральных свойств. Сложные вычисления и работа с параметрами выполняются вне ядра — в его обвязке. В связи с этим обвязка ядра имеет гораздо большее пространство состояний, чем само ядро. Это позволяет проводить верификацию ядра методом проверки модели (model checking) с минимальными затратами вычислительных ресурсов. Более того, небольшое пространство состояний ядра при использовании подходящих инструментов позволяет выполнять автоматическое извлечение модели для верификации из кода программы, не прибегая к каким-либо абстракциям.

Для верификации модели с учётом симметрии требуется специальная инструментальная поддержка. Поведение ядра представляет собой уже упрощённую с учётом симметрии модель, поэтому для верификации может быть использован любой инструмент проверки модели.

Потенциально предложенный подход позволит выполнять верификацию достаточно сложных программ, содержащих какой-либо вид симметрии. Это требует от разработчика понимания присущей программе симметрии, так как она должна быть заложена в ядро на этапе проектирования. Достоинством подхода является то, что не требуется определять группу симметрии и строить на её основе фактормодель для верификации.

Приведённый пример разработки и верификации арбитра ресурсов марсохода демонстрирует практическую ценность результатов настоящей работы.

References

[1] D. Harel and A. Pnueli, "On the Development of Reactive Systems", in *Logics and Models of Concurrent Systems*, vol. 13, 1985, pp. 477–498. DOI: 10.1007/978-3-642-82453-1_17.

[2] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends", in *Current Trends in Concurrency*, vol. 224, 1986, pp. 510–584. DOI: 10.1007/BFb0027047.

[3] K. Schneider, J. Shabolt, and J. G. Taylor, *Verification of Reactive Systems: Formal Methods and Algorithms* (Texts in Theoretical Computer Science), 1st edition. Springer, 2004, ISBN: 978-3-540-00296-3. DOI: 10.1007/978-3-662-10778-2.

[4] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*, 1st edition. Springer, 2018, p. 1212, ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8.

[5] E. M. Clarke, O. Grumberg, and D. Peled, *Verification of Program Models: Model Checking*. MCNMO, 2002, p. 416, translated from English to Russian, ISBN: 5-94057-054-2.

[6] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008, p. 975, ISBN: 978-0-262-02649-9.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the State Explosion Problem in Model Checking", in *Informatics: 10 Years Back, 10 Years Ahead*, ser. LNCS, vol. 2000, 2001, pp. 176–194. DOI: 10.1007/3-540-44577-3_12.

[8] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla, "Symmetry Reductions in Model Checking", in *Computer Aided Verification*, 1998, pp. 147–158. DOI: 10.1007/BFb0028741.

[9] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting Symmetry in Temporal Logic Model Checking", *Formal Methods in System Design*, vol. 9, no. 1–2, pp. 77–104, 1996. DOI: 10.1007/BF00625969.

[10] A. Miller, A. Donaldson, and M. Calder, "Symmetry in Temporal Logic Model Checking", *ACM Computing Surveys*, vol. 38, no. 3, pp. 1–36, 2006. DOI: 10.1145/1132960.1132962.

[11] E. A. Emerson and A. P. Sistla, "Symmetry and Model Checking", *Formal Methods in System Design*, vol. 9, no. 1, pp. 105–131, 1996. DOI: 10.1007/BF00625970.

[12] S. Barner and O. Grumberg, "Combining Symmetry Reduction and Under-Approximation for Symbolic Model Checking", in *International Conference on Computer Aided Verification*, 2002, pp. 93–106. DOI: 10.1007/3-540-45657-0_8.

[13] D. Tang, S. Malik, A. Gupta, and C. N. Ip, "Symmetry Reduction in SAT-Based Model Checking", in *International Conference on Computer Aided Verification*, ser. LNCS, vol. 3576, 2005, pp. 125–138. DOI: 10.1007/11513988_12.

[14] A. P. Sistla, "Employing Symmetry Reductions in Model Checking", *Computer Languages, Systems & Structures*, vol. 30, no. 3-4, pp. 99–137, 2004. DOI: 10.1016/j.cl.2004.02.002.

[15] C. N. Ip and D. L. Dill, "Better Verification Through Symmetry", in *Computer Hardware Description Languages and their Applications*, 1993, pp. 97–111. DOI: 10.1016/B978-0-444-81641-2.50012-5.

[16] S. J. Zhang, J. Sun, C. Sun, Y. Liu, J. Ma, and J. S. Dong, *Symmetry Detection for Model Checking*, 2013.

[17] M. Leuschel and T. Massart, "Efficient Approximate Verification of B and Z Models via Symmetry Markers", *Annals of Mathematics and Artificial Intelligence*, vol. 59, no. 1, pp. 81–106, 2010. DOI: 10.1007/s10472-010-9208-8.

[18] A. P. Sistla, V. Gyuris, and E. A. Emerson, "SMC: A Symmetry-Based Model Checker for Verification of Safety and Liveness Properties", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 2, pp. 133–166, 2000. DOI: 10.1145/350887.350891.

[19] A. P. Sistla and P. Godefroid, "Symmetry and Reduced Symmetry in Model Checking", *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 4, pp. 702–734, 2004. DOI: 10.1145/1011508.1011511.

[20] A. F. Donaldson and A. Miller, "Automatic Symmetry Detection for Model Checking Using Computational Group Theory", in *International Symposium on Formal Methods*, 2005, pp. 481–496. DOI: 10.1007/11526841_32.

[21] I. Valkov, A. F. Donaldson, and A. Miller, "Synchronisation in Language-Level Symmetry Reduction for Probabilistic Model Checking", in *International Symposium on Model Checking Software*, Springer, 2024, pp. 49–66. DOI: 10.1007/978-3-031-66149-5_3.

[22] A. F. Donaldson and A. Miller, "Exact and Approximate Strategies for Symmetry Reduction in Model Checking", in *International Symposium on Formal Methods*, 2006, pp. 541–556. DOI: 10.1007/11813040_36.

[23] T. Gibson-Robinson and G. Lowe, "Symmetry Reduction in CSP Model Checking", *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 5, pp. 567–605, 2019. DOI: 10.1007/s10009-019-00516-4.

[24] M. M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, and A. Movaghar, "Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca", *Acta Informatica*, vol. 47, no. 1, pp. 33–66, 2010. DOI: 10.1007/s00236-009-0111-x.

[25] M. Leuschel, M. Butler, C. Spermann, and E. Turner, "Symmetry Reduction for B by Permutation Flooding", in *International Conference of B Users*, vol. 4355, 2007, pp. 79–93. DOI: 10.1007/11955757_9.

[26] L. Antuña, D. Araiza-Illan, S. Campos, and K. Eder, "Symmetry Reduction Enables Model Checking of More Complex Emergent Behaviours of Swarm Navigation Algorithms", in *Towards Autonomous Robotic Systems*, vol. 9287, 2015, pp. 26–37. DOI: 10.1007/978-3-319-22416-9_4.

[27] I. Buzhinsky and A. Pakonen, "Model-Checking Detailed Fault-Tolerant Nuclear Power Plant Safety Functions", *IEEE Access*, vol. 7, 162139–162156, 2019. DOI: 10.1109/ACCESS.2019.2951938.

[28] I. Buzhinsky and A. Pakonen, "Symmetry Breaking in Model Checking of Fault-Tolerant Nuclear Instrumentation and Control Systems", *IEEE Access*, vol. 8, 197684–197694, 2020. DOI: 10.1109/ACCESS.2020.3034799.

[29] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic", in *25 Years of Model Checking: History, Achievements, Perspectives*, ser. LNCS, vol. 5000, 2008, pp. 196–215, ISBN: 978-3-540-69849-4. DOI: 10.1007/978-3-540-69850-0_12.

[30] D. Park, “Concurrency and Automata on Infinite Sequences”, in *Theoretical Computer Science: 5th GI-Conference Karlsruhe*, vol. 104, 1981, pp. 167–183. DOI: 10.1007/BFb0017309.

[31] G. J. Holzmann and R. Joshi, “Model-Driven Software Verification”, in *International SPIN Workshop on Model Checking of Software*, Springer, vol. 2989, 2004, pp. 76–91. DOI: 10.1007/978-3-540-24732-6_6.

[32] V. M. Glushkov, *The Synthesis of Digital Automata*. Moscow: PhisMathGis, 1962, p. 476, in Russian.

[33] M. V. Neyzov and E. V. Kuzmin, “LTL-Specification for Development and Verification of Logical Control Programs in Feedback Systems”, *Modeling and Analysis of Information Systems*, vol. 31, no. 3, pp. 240–279, 2024. DOI: 10.18255/1818-1015-2024-3-240-279.

[34] G. Holzmann, *SPIN Model Checker: The Primer and Reference Manual*. Addison-Wesley, 2003, p. 608, ISBN: 0-321-22862-6.