Модел. и анализ информ. систем. Т. **16**, № **1** (2009) 92–112

УДК 004.415.52

Раздельная верификация объектно-ориентированных программ с построением протокола C++ класса в терминах сетей Петри

Харитонов Д.И.[1]

Институт автоматики и процессов управления Дальневосточного отделения РАН

e-mail: demiurg@dvo.ru

получена 12 марта 2009

Ключевые слова: языки программирования, формальная верификация программ, объектно-ориентированное программирование, сети Петри

Формальное доказательство корректности программ является единственным надежным методом обеспечения правильного функционирования программных систем. Существующие методы верификации программ на практике довольно сложны, и это препятствует их повсеместному применению. В настоящей статье предлагается подход, применяющий концепцию "Design By Contract", в котором протоколы классов, сформулированные в терминах сетей Петри, позволяют проводить анализ классов и программ раздельно, что облегчает проверку работоспособности объектно-ориентированных программ.

1. Введение

Проверка корректности программ – это сложная, многоплановая математическая задача, не имеющая в общем случае алгоритмического решения. С одной стороны, под корректностью программы подразумевается выполнение поставленной перед ней задачи, т.е. корректность выбора и реализации алгоритма. С другой стороны, корректность программы означает работоспособность программы, то есть отсутствие в ней ошибок, приводящих к зацикливанию, аварийным остановкам и бесконечному ожиданию. Доказательство корректности программ называется верификацией. Методы и средства верификации программ разрабатываются на протяжении уже многих десятилетий. Стоит упомянуть одну из первых работ Хоара [10], заложившую в 1969 году основы целого теоретического направления математической верификации программ. Разработка качественных средств верификации программ занимает много времени и становится практически делом всей жизни [6, 7]. Однако

[1]Работа выполнена по программам Президиума РАН № 1 и № 2

верификация пока ещё не является обязательной частью процесса разработки программ. В большинстве случаев вместо верификации программ применяется процесс тестирования и отладки [8, 5], способный выявить ошибки, приводящие к неработоспособности программы. Однако с увеличением сложности разрабатываемых программных продуктов стало всё более очевидным, что тестирование не в состоянии полностью решить эту задачу. Подтверждением этому является множество исправляемых ежегодно DOS[2] ошибок в выпущенном в эксплуатацию серверном программном обеспечении.

Сегодня остаётся верным утверждение почти что двадцатилетней давности из [4] – *основными препятствиями применения методов верификации обычно являются не принципиальные (теоретические) ограничения, а несовершенство традиционных технологий и отсутствие подготовленных кадров, владеющих соответствующим аппаратом и методикой, а также инструментальных средств поддержки верификации.* К сожалению, за прошедшее время уровень подготовки специалистов не вырос, а скорее даже наоборот понизился и, следовательно, единственным способом преодоления препятствий является разработка новых методов и инструментальных средств верификации, упрощающих традиционный технологический процесс верификации.

Одним из перспективных подходов к разработке новых методов верификации программ является применение формальной теории сетей Петри [11]. Во-первых, сети Петри уже доказали на практике свою состоятельность в области разработки коммуникационных протоколов [1, 3], которые являются узковыделенной областью программирования. А во-вторых, некорректности программ имеют прямую интерпретацию в сетях Петри: ошибки, приводящие к бесконечному ожиданию – это дедлоки или тупики; зацикливание программы – это отсутствие живости переходов; а аварийная остановка – это недостижимость конечного состояния или достижимость аварийных состояний.

Общую схему описания и анализа протоколов в терминах сетей Петри можно представить следующим образом [1]. Из неформального описания протокола строятся в терминах сетей Петри описания протокольных объектов и среды, которые вместе составляют модель реализации протокола. Эта модель анализируется на свойства живости и ограниченности (безопасности). Применительно к программированию подобный анализ можно сравнить с проверкой работоспособности программ. При наличии модели сервиса протокола, которая отличается от модели реализации отсутствием детализации, возможно также сравнение этих моделей на соответствие реализации желаемому сервису. Применительно к программированию это можно назвать проверкой корректности реализации алгоритма.

В настоящей статье автор демонстрирует, что схема описания и анализа протоколов применима при разработке и анализе объектно-ориентированных программ. При этом в качестве аналога взаимодействующих сторон протоколов выступают экземпляры классов, взаимодействующие посредством вызова функций и обращения к переменным членам классов. А в качестве протоколов – правила вызовов функций и обращения к переменным, которые в настоящее время существуют в

[2]DOS - Deny of service.

виде неформальных описаний классов, сделанных разработчиками. Применение сетей Петри при формализации описаний протоколов класса позволяет использовать эти описания для раздельного анализа корректности реализации класса и проверки корректности обращений к классу методами, наработанными в области теории протоколов. Тем самым проблема верификации программ разделяется во времени по мере разработки классов и их использования в программе.

Далее в статье будет использован следующий подход для описания метода проверки работоспособности объектно-ориентированных программ. После введения основных определений и терминов (раздел 2.) в разделе 3. будет введено отношение частичного порядка на сетях Петри, позволяющее сравнивать поведение помеченных сетей Петри. В разделе 4. будет описан метод проверки работоспособности объектно-ориентированных программ с применением введённого в настоящей статье отношения частичного порядка между сетями Петри. В разделе 5. приводится пример проверки работоспособности класса и программы, использующей этот класс, с применением предлагаемого метода. В заключение статьи проводится сравнение предложенного метода с другими методами верификации программ и отмечаются возможные пути его развития.

2. Основные понятия

Пусть $A = \{a_1, a_2, ..., a_k\}$ есть некоторое множество. Мультимножеством на множестве A есть функция $\mu : A \to 0, 1, 2, ...$, которая ассоциирует с каждым элементом множества A неотрицательное целое число. Иногда удобно записывать мультимножество на множестве A как формальную сумму $n_1a_1 + n_2a_2 + ... + n_ka_k$ или $\Sigma n_i a_i$, где $n_i = \mu(a_i)$ есть число вхождений $a_i \in A$ в мультимножество. Как правило, при записи суммы её элементы с $n_i = 0$ будут опускаться. Объединение и пересечение двух мультимножеств $\mu_1 = n_1a_1 + n_2a_2 + ... + n_ka_k$ и $\mu_2 = m_1a_1 + m_2a_2 + ... + m_ka_k$ на множестве A определяется соответственно как $\mu_1 + \mu_2 = (n_1 + m_1)a_1 + (n_2 + m_2)a_2 + ... + (n_k + m_k)a_k$ и $\mu_1 - \mu_2 = (n_1 - m_1)a_1 + (n_2 - m_2)a_2 + ... + (n_k - m_k)a_k$, где последняя операция производится только тогда, когда $n_i > m_i$ для всех $1 \leq i \geq k$. Мы будем писать $\mu_1 \leq \mu_2$, если $n_i \leq m_i$ для каждого $1 \leq i \geq k$, и $\mu_1 < \mu_2$, если $\mu_1 \leq \mu_2$ и $\mu_1 \neq \mu_2$. Если $n_i = 0$ для всех i, тогда такое мультимножество будет обозначаться как $\mathbf{0}$. Мы будем также писать, что $a \in \mu$, если $\exists n > 0 : (a, n) \in \mu$.

Множество всех конечных мультимножеств на множестве A будет обозначаться как $\mathcal{M}(A)$. Множество всех конечных последовательностей, составленное из символов множества A, включая пустую строку ϵ, будет обозначаться как A^*.

Определение 1. *Сеть Петри*

Сеть Петри определяется как набор $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet, M_0 \rangle$, где

1. S есть конечное множество *мест*;
2. T есть конечное множество *переходов* такое, что $S \cap T = \emptyset$;
3. ${}^\bullet() : T \to \mathcal{M}(S)$: есть входная функция инцидентности;
4. $()^\bullet : T \to \mathcal{M}(S)$: есть выходная функция инцидентности;

5. $M_0 \in \mathcal{M}(S)$ есть начальная маркировка.

■ 1

Мультимножества ${}^\bullet t$ и $t^\bullet$ называются входными и выходными множествами перехода $t \in T$ соответственно. Функции ${}^\bullet()$ и $()^\bullet$ могут быть естественно расширены на мультимножества ${}^\bullet(n_1t_1 + n_2t_2 + ... + n_kt_k) = n_1{}^\bullet t_1 + n_2{}^\bullet t_2 + ... + n_k{}^\bullet t_k$ и $(n_1t_1 + n_2t_2 + ... + n_kt_k)^\bullet = n_1t_1^\bullet + n_2t_2^\bullet + ... + n_kt_k^\bullet$.

Далее будет использоваться привычное графическое представление сети Петри как двудольного графа, где места изображаются кружками, а переходы - прямоугольниками. Места и переходы соединяются ориентированными дугами, представляя входные и выходные функции инцидентности. Вес дуг задаётся целыми числами, расположенными около дуг. Маркировка изображается метками, помещёнными внутрь мест.

В данной работе будет использоваться шаговая семантика сетей Петри, основанная на срабатывании последовательностей мультимножеств переходов.

Маркировка M сети $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet, M_0\rangle$ есть мультимножество, заданное на S, т.е. $M \in \mathcal{M}(S)$. Будем говорить, что шаг (мультимножество переходов) $\Theta \in \mathcal{M}(T)$ возбуждён в маркировке M, если ${}^\bullet\Theta \leq M$. Шаг $\Theta \in \mathcal{M}(T)$, возбуждённый в маркировке M, может сработать, приводя к новой маркировке M', что записывается как $M[\Theta\rangle M'$, где $M' = M -^\bullet \Theta + \Theta^\bullet$. Следует отметить, что, если шаг Θ возбуждён в маркировке M, то шаг $\Theta' < \Theta$ также возбуждён в маркировке M. Если $\Phi = \Theta_1\Theta_2...\Theta_n \in (\mathcal{M}(T))^*$, то $M[\Phi\rangle M'$ означает, что существует последовательность $M_1M_2...M_{n-1}$ такая, что $M[\Theta_1\rangle M_1[\Theta_2\rangle ...M_{n-1}[\Theta_n\rangle M'$. В этом случае будем говорить, что M' достижима из M. $M[\rangle M'$ означает, что существует $\Phi \in (\mathcal{M}(T))^*$ такая, что $M[\Phi\rangle M'$. Множество всех достижимых маркировок из M определяется как $[M\rangle = \{M' | M[\rangle M'\}$. Если мы желаем указать сеть, для которой формулируется некоторое утверждение, мы будем указывать соответствующий префикс. Например, $\Sigma : M[\Phi\rangle M', \Sigma_1 : M \leq^\bullet \Theta$.

Пусть Δ есть конечный алфавит имён, и $\overline{\Delta} = \{\overline{a} \mid a \in \Delta\}$ есть связанный с ним алфавит дополнительных имён. Другими словами, мы определяем биекцию $\bar{\cdot} : \Delta \to \overline{\Delta}$, которая означает взаимноднозначное соответствие между именами и их дополнительными именами. Для простоты функция, обратная функции $\bar{\cdot}$, будет записываться как $\bar{\cdot} : \overline{\Delta} \to \Delta$. Таким образом, мы имеем $\overline{\overline{a}} = a$. Пусть $\tau \notin \mathcal{M}(\Delta \cup \overline{\Delta})$ есть специальный символ, который ассоциируется с невидимым действием. Далее будем обозначать $Vis = \Delta \cup \overline{\Delta}$ и $Act = \mathcal{M}(Vis) \cup \{\tau\}$. Функция $\bar{\cdot}$ может быть расширена на мультимножество имён $\overline{n_1a_1 + ...n_ka_k} = n_1\overline{a_1} + ...n_k\overline{a_k}$. Например, $\overline{\overline{a} + 2a + 3\overline{b} + c} = a + 2\overline{a} + 3b + \overline{c}$.

Определение 2. *Пометка*

Пометка сети Петри $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet, M_0\rangle$ есть набор $\lambda = \langle \Delta, \sigma\rangle$, где Δ есть некоторый алфавит, и $\sigma : T \to Act$ есть пометочная функция. ■ 2

Это определение является расширением хорошо известного определения пометочной функции. В частности, каждый переход может быть помечен не только одним символом, но и их мультимножеством. Переход, помеченный символом τ, будет трактоваться как невидимый или внутренний переход.

Физическим смыслом метки является элементарное взаимодействие. Так, например, будем считать, что символ $a \in \Delta$ соответствует посылке сообщения с именем a, а дополнительный символ $\overline{a} \in \overline{\Delta}$ - приёму сообщения с именем a. Символ τ не связан с взаимодействием и характеризует некоторое внутреннее событие. Мультимножество символов, помечающее переход, соответствует одновременному приёму и/или передаче сообщений, происходящих при срабатывании этого перехода. Так, например, если $\sigma(t) = \overline{a} + 2b$, то срабатыванию перехода t соответствует одновременный приём сообщения a и посылка двух сообщений b.

Пометочная функция σ расширяется на мультимножество переходов, $\sigma : \mathcal{M}(T) \to Act$, следующим образом. Если $\Theta = \Sigma n_i t_i \in \mathcal{M}(T)$, то $\sigma(\Theta) = \Sigma n_i \sigma(t_i)$. Если каждый переход t в шаге Θ невидим, т.е. $\sigma(t) = \tau$, тогда мы будем писать $\sigma(\Theta) = \tau$.

Функция σ может быть также расширена до гомоморфизма $\sigma : (\mathcal{M}(T))^* \to (\mathcal{M}(Act))^*$. Определим функцию $\sigma^+ : (\mathcal{M}(T))^* \to (Vis)^*$, которая удаляет все τ символы из последовательностей:

$$\sigma^+(\Theta) = \begin{cases} \epsilon, & \text{если } \sigma(\Theta) = \tau; \\ \sigma(\Theta), & \text{в противном случае;} \end{cases}$$

$$\sigma^+(\Phi\Theta) = \sigma^+(\Phi) + \sigma^+(\Theta).$$

Далее буквой Θ будет обозначаться шаг, а буквой Φ – последовательность шагов.

Если $W \in (\mathcal{M}(Vis))^*$ и $\lambda = \langle \Delta_\lambda, \sigma_\lambda \rangle$ – пометки сети Петри Σ, тогда $M(W\rangle_\lambda M'$ означает, что $\exists \Phi \in (\mathcal{M}(T))^* : M[\Phi\rangle M'$ и $\sigma^+_\lambda(\Phi) = W$. Далее для краткости последовательности $M[\Phi\rangle M'$ с $\sigma(\Phi) = \epsilon$ будет записываться как $M \Rightarrow M'$, а сам символ $\Rightarrow$ будет обозначать невидимую последовательность шагов.

Определение 3. *Ограничение.*

Пусть дана сеть $\Sigma_1 = \langle S_1, T_1, {}^\bullet()_1, ()^\bullet_1, M_{01} \rangle$ и её пометка $\alpha = \langle \Delta_\alpha, \sigma_\alpha \rangle$. Тогда ограничение сети Σ по отношению к α есть новая сеть $\Sigma = \partial_\alpha(\Sigma_1)$ такая, что

1. $S = S_1$;
2. $T = T_1 \backslash \{t \in T | \sigma_\alpha(t) \neq \tau\}$;
3. ${}^\bullet(t) = {}^\bullet(t)_1, t \in T$;
4. $(t)^\bullet = (t)^\bullet_1, t \in T$;
5. $M_0 = M_{01}$.

■ 3

Менее формально, ограничение сети удаляет каждый переход, помеченный именем из Vis, вместе с прилегающими дугами.

Предположим, что для сети Σ_1 определены две пометки $\alpha = \langle \Delta_\alpha, \sigma_\alpha \rangle$ и $\beta = \langle \Delta_\beta, \sigma_\beta \rangle$. Пометка β для сети $\Sigma = \partial_\alpha(\Sigma_1)$ может быть естественно ограничена следующим образом: $\tilde{\beta} = \langle \Delta_\beta, \sigma_\beta \lceil T \rangle$. Иногда, когда это не приводит к недоразумениям, будет записываться β вместо $\tilde{\beta}$. Результат последовательного применения к сети Σ_1 ограничения по пометкам α и β будем записывать как $\partial_{\{\alpha,\tilde{\beta}\}}(\Sigma_1) \equiv \partial_{\tilde{\beta}}(\partial_\alpha(\Sigma_1))$.

Пусть $\Sigma_1 = \langle S_1, T_1, {}^\bullet()_1, ()^\bullet_1, M_{01} \rangle$ и $\Sigma_2 = \langle S_2, T_2, {}^\bullet()_2, ()^\bullet_2, M_{02} \rangle$ – две сети Петри. Пусть также сети Σ_1 и Σ_2 имеют непересекающиеся множества мест и переходов, т.е. $S_1 \cap S_2 = T_1 \cap T_2 = \emptyset$.

Определение 4. *Параллельная композиция мультипомеченных сетей*

Пусть Σ_1 и Σ_2 – сети Петри с пометками $\alpha = \langle \Delta_\alpha, \sigma_\alpha \rangle$ и $\beta = \langle \Delta_\beta, \sigma_\beta \rangle$ соответственно. Параллельная композиция сетей Σ_1 и Σ_2 по отношению к α и β есть новая сеть $\Sigma = (\Sigma_1 \ {}_\alpha|_\beta \ \Sigma_2)$ такая, что

1. $S = S_1 \cup S_2$,
2. $T = T_1 \cup T_2 \cup T_1 \ {}_\alpha \otimes_\beta T_2$, где
   $T_1 \ {}_\alpha \otimes_\beta T_2 = \{\mu_1 + \mu_2 \mid \mu_1 \in \mathcal{M}(T_1), \mu_2 \in \mathcal{M}(T_2), \tau \notin \sigma_\alpha(\mu_1) = \overline{\sigma_\beta(\mu_2)},$ сумма $\mu_1 + \mu_2$ минимальна$\}$.
3. ${}^\bullet() = {}^\bullet()_1 \cup {}^\bullet()_2 \cup \{(\mu_1 + \mu_2, \ {}^\bullet(\mu_1)_1 \ + \ {}^\bullet(\mu_2)_2) \mid \mu_1 + \mu_2 \in T, \mu_1 \in \mathcal{M}(T_1), \mu_2 \in \mathcal{M}(T_2)\}$,
4. $()^\bullet = \ ()_1^\bullet \ \cup \ ()_2^\bullet \ \cup \ \{(\mu_1 + \mu_2, (\mu_1)_1^\bullet \ + \ (\mu_2)_2^\bullet) \mid \mu_1 + \mu_2 \in T, \mu_1 \in \mathcal{M}(T_1), \mu_2 \in \mathcal{M}(T_2)\}$,
5. $M_0 = M_{01} + M_{02}$.

Сумма $\mu_1 + \mu_2$ минимальна, если не существует суммы $\mu_1' + \mu_2'$ такой, что $\mu_1' + \mu_2' < \mu_1 + \mu_2$ и $\sigma_\alpha(\mu_1') = \overline{\sigma_\beta(\mu_2')}$. ■ 4

Менее формально, две сети Σ_1 и Σ_2 просто объединяются с добавлением новых переходов синхронизации $T_1 \ {}_\alpha \otimes_\beta T_2$. Эти новые переходы задаются мультимножеством символов $\mu_1 + \mu_2, \mu_1 \in \mathcal{M}(T_1), \mu_2 \in \mathcal{M}(T_2)$. Для новых переходов вычисляются их входные и выходные мультимножества: ${}^\bullet(\mu_1 + \mu_2) = {}^\bullet(\mu_1) \ + \ {}^\bullet(\mu_2), (\mu_1 + \mu_2)^\bullet = (\mu_1)^\bullet \ + \ (\mu_2)^\bullet$.

Определение 5. *Синхронизация мультипомеченных сетей*

Пусть Σ_1 и Σ_2 сети Петри с пометками $\alpha = \langle \Delta_\alpha, \sigma_\alpha \rangle$ и $\beta = \langle \Delta_\beta, \sigma_\beta \rangle$ соответственно. Синхронизацией сетей Σ_1 и Σ_2 по отношению к α и β назовём новую сеть $\Sigma = \partial_{\{\alpha, \tilde{\beta}\}}(\Sigma_1 \ {}_\alpha|_\beta \ \Sigma_2)$. Будем также считать, что пометки α и β, по которым производилась синхронизация, доопределены для сети Σ следующим образом: $\forall t \in T_1 \cup T_2 \Longrightarrow \sigma_\alpha(t) = \emptyset, \sigma_\beta(t) = \emptyset$, $\forall t = \mu_1 + \mu_2, \mu_1 \in \mathcal{M}(T_1), \mu_2 \in \mathcal{M}(T_2) \Longrightarrow \sigma_\alpha(t) = \sigma_\alpha(\mu_1), \sigma_\beta(t) = \sigma_\beta(\mu_2)$.

Обозначим результат такой композиции как $\Sigma = (\Sigma_1 {}_\alpha \|_\beta \Sigma_2)$, а в случае, когда это не приводит к недоразумениям, будем записывать более коротко $\Sigma = (\Sigma_1 \| \Sigma_2)$.
■ 5

Результат синхронизации мультипомеченных сетей получается после выполнения трёх шагов. Во-первых, выполняется композиция сетей. Во-вторых, из сети, являющейся результатом композиции, удаляются исходные помеченные переходы. В-третьих, для переходов синхронизации, добавленных при построении композиции исходных сетей, доопределяются пометочные функции.

Введём отношения, позволяющие сравнивать поведение помеченных сетей Петри.

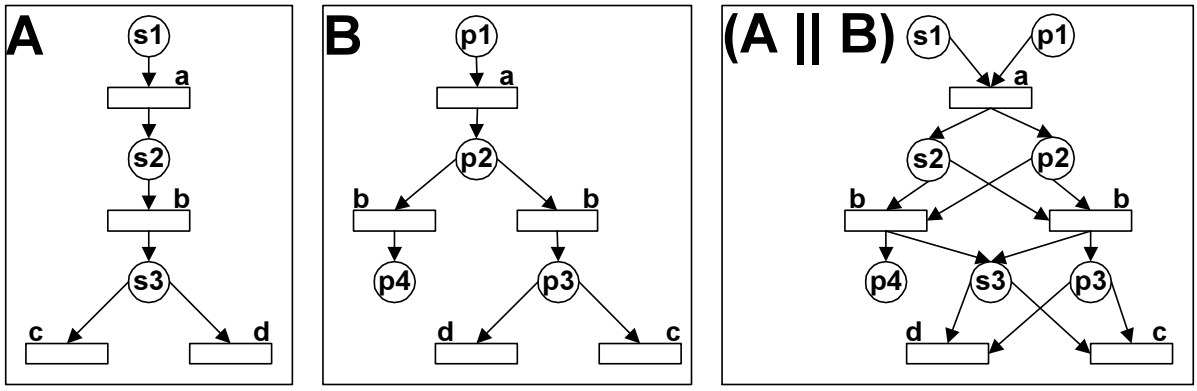

Рис. 1. Простые сети Петри и результат их композиции

Определение 6. *Бисимуляционная эквивалентность помеченных сетей Петри.*

Пусть A и B – сети Петри с пометками a и b. Будем говорить, что сети Петри A и B бисимуляционно эквивалентны, и записывать $(A,a) \approx^{\mathfrak{R}} (B,b)$, если существует отношение бисимуляции $\mathfrak{R} \subseteq [M_{0A}\rangle \times [M_{0B}\rangle$ такое, что

1. $(M_{0A}, M_{0B}) \in \mathfrak{R}$.
2. если $(M_A, M_B) \in \mathfrak{R}$ и $W \in (\mathcal{M}(Vis))^*$, то
   (a) если $M_A(W\rangle_a M'_A$, то $\exists M'_B : M_B(W\rangle_b M'_B$ и $(M'_A, M'_B) \in \mathfrak{R}$,
   эквивалентные условия в одношаговой версии определения:
   если $M_A[\Theta_A\rangle_a M'_A$, то $\exists \Phi_B \in (\mathcal{M}(T_B))^* : M_B[\Phi_B\rangle_b M'_B$, $(M'_A, M'_B) \in \mathfrak{R}$ и $\sigma_a(\Theta_A) = \sigma_b(\Phi_B)$,
   (b) если $M_B(W\rangle_b M'_B$, то $\exists M'_A : M_A(W\rangle_a M'_A$ и $(M'_A, M'_B) \in \mathfrak{R}$,
   эквивалентные условия в одношаговой версии определения:
   если $M_B[\Theta_B\rangle_b M'_B$, то $\exists \Phi_A \in (\mathcal{M}(T_A))^* : M_A[\Phi_A\rangle_a M'_A$, $(M'_A, M'_B) \in \mathfrak{R}$, и $\sigma_b(\Theta_B) = \sigma_a(\Phi_A)$.

Если для отношения $\mathfrak{R}$ выполняются условия 1) и 2-a), то будем говорить, что A содержит все события сети B или что поведение сети частично вкладывается в поведение сети A: $(A,a) \gtrsim^{\mathfrak{R}} (B,b)$. Если выполняются условия 1) и 2-b), то будем говорить, что B содержит все события сети A: $(A,a) \lesssim^{\mathfrak{R}} (B,b)$. ■ 6

Утверждение 7. *Вложенность событий синхронизации сетей Петри*

Пусть A и B - сети Петри с пометками $\alpha = \langle \Delta_\alpha, \sigma_\alpha \rangle$ и $\beta = \langle \Delta_\beta, \sigma_\beta \rangle$ соответственно. Сеть $N_{AB} = (A \,{}_\alpha\|_\beta\, B)$ - результат синхронизации сетей A и B.

Тогда существуют отношения $\mathfrak{R}_{aa}$ и $\mathfrak{R}_{bb}$ такие, что $(A,\alpha) \gtrsim^{\mathfrak{R}_{aa}} (N_{AB},\alpha)$, $(B,\beta) \gtrsim^{\mathfrak{R}_{bb}} (N_{AB},\beta)$. ■ 7

Док-во: По правилам построения сети N_{AB} как результата синхронизации сетей A и B:
1. $M_{0AB} = M_{0A} + M_{0B}$.
2. $M_{1AB} = M_{1A} + M_{1B}, M_{1AB}(W\rangle_{\alpha'} M_{2AB} \Longrightarrow$
$M_{2AB} = M_{2A} + M_{2B},\ M_{1A}(W\rangle_{\alpha} M_{2A},\ M_{1B}(\overline{W}\rangle_{\beta} M_{2B}$. ■

3. Сравнение поведения помеченных сетей Петри

Отношение бисимуляционной эквивалентности обладает свойствами симметричности и транзитивности. Отношение вложенности событий содержит в себе половину условий, необходимых для бисимуляционной эквивалентности сетей Петри, благодаря этому оно обладает свойством транзитивности. На рисунке 1 приведён пример сетей A и B, между которыми существуют отношения вложенности событий $(A,\alpha) \gtrsim^{\mathfrak{R}_1} (B,\beta)$[3] и $(A,\alpha) \lesssim^{\mathfrak{R}_2} (B,\beta)$[4], однако отношение $(A,\alpha) \approx^{\mathfrak{R}} (B,\beta)$ не выполняется. То есть отношение вложенности событий не обладает ни свойством симметричности, ни свойством антисимметричности, необходимыми, как будет показано в дальнейшем, для метода верификации программ.

Введём новый тип отношения между помеченными сетями Петри, устанавливающий однозначный порядок между сетями на рисунке 1.

Определение 8. *Отношение "допускает" между простыми сетями Петри*

Пусть A и B - простые сети Петри с пометками $\alpha = \langle \Delta_\alpha, \sigma_\alpha \rangle$ и $\beta = \langle \Delta_\beta, \sigma_\beta \rangle$ соответственно. $N_{AB} = (A \,{}_{\overline{\alpha}}\|_{\beta}\, B)$ - сеть Петри, полученная в результате синхронизации сетей A и B по пометкам $\overline{\alpha}$ и β.

Тогда будем говорить, что сеть A *допускает* B , или что поведение сети полностью вкладывается в поведение сети A, и записывать $(A,\alpha) \succ (B,\beta)$ тогда и только тогда, когда сеть Петри N_{AB} бисимуляционно эквивалентна сети B : $(N_{AB},\beta) \approx^{\mathfrak{R}} (B,\beta)$. ■ 8

В приведённом на рисунке 1 примере между сетями A и B, кроме упомянутых ранее отношений вложенности событий, выполняется также и отношение $(A,\alpha) \succ (B,\beta)$. Однако отношение $(B,\beta) \succ (A,\alpha)$ не выполняется, так как в сети A после срабатывания перехода, помеченного символом b, обязательно идёт срабатывание переходов с видимостью c или d, а в сети B из состояния p_4 таких срабатываний нет.

Утверждение 9. *О вложенности событий, определяемых отношением "допускает"*

Пусть A и B - простые сети Петри с пометками $\alpha = \langle \Delta_\alpha, \sigma_\alpha \rangle$ и $\beta = \langle \Delta_\beta, \sigma_\beta \rangle$ соответственно. $N_{AB} = (A \,{}_{\overline{\alpha}}\|_{\beta}\, B)$ – сеть Петри, полученная в результате синхронизации сетей A и B по пометкам $\overline{\alpha}$ и β. Тогда если $(A,\alpha) \succ (B,\beta)$, то :
1. $\exists \mathfrak{R}_{bb} : (N_{AB},\beta) \gtrsim^{\mathfrak{R}_{bb}} (B,\beta)$

[3] $\mathfrak{R}_1 = \{(s_1,p_1),(s_2,p_2),(s_3,p_4),(\emptyset,\emptyset)\}$
[4] $\mathfrak{R}_2 = \{(p_1,s_1),(p_2,s_2),(p_3,s_3),(p_4,s_3),(\emptyset,\emptyset)\}$

2. $\exists \mathfrak{R}_{ab} : (A, \overline{\alpha}) \gtrsim^{\mathfrak{R}_{ab}} (B, \beta)$
3. $\exists \mathfrak{R} : (N_{AB}, \beta) \approx^{\mathfrak{R}} (B, \beta) : \forall (M_{AB}, M_B) \in \mathfrak{R} \Longleftrightarrow M_{AB} = M_A + M_B$ ■ 9

Док-во: Построим необходимые для выполнения условий 1-3 бинарные отношения.
Пусть $\mathfrak{R}'$ – отношение бисимуляции : $(N_{AB}, \beta) \approx^{\mathfrak{R}'} (B, \beta)$, тогда
1. $\mathfrak{R}_{bb} = \mathfrak{R}'$
2. $\mathfrak{R}_{ab} = \{(M_A, M_B) \mid (M_A + M'_B, M_B) \in \mathfrak{R}'\}$
3. $\mathfrak{R} = \{(M_A + M_B, M_B) \mid \exists M'_B : (M_A + M'_B, M_B) \in \mathfrak{R}'$ или $(M_A + M_B, M'_B) \in \mathfrak{R}'\}$ ■

Утверждение 10. *Транзитивность отношения "допускает"*

Пусть A, B, C – простые сети Петри с пометками $\alpha = \langle \Delta_\alpha, \sigma_\alpha \rangle$, $\beta = \langle \Delta_\beta, \sigma_\beta \rangle$ и $\gamma = \langle \Delta_\gamma, \sigma_\gamma \rangle$ соответственно. Если между сетями установлены отношения $A \,^{\alpha} \succ^{\beta} B$, $B \,^{\beta} \succ^{\gamma} C$, то $A \,^{\alpha} \succ^{\gamma} C$. ■ 10

Док-во: Из определения отношения *допускает*:
$\exists \mathfrak{R}_1 = \{(M_B, M_B + M_B)\} \subseteq [M_{0B}\rangle \times [M_{0AB}\rangle : (B, \beta) \approx^{\mathfrak{R}_1} (N_{AB}, \beta)$, где $N_{AB} = (A \,{}_{\overline{\alpha}}\|_{\beta}\, B)$.
$\exists \mathfrak{R}_2 = \{(M_C, M_C + M_B)\} \subseteq [M_{0C}\rangle \times [M_{0BC}\rangle : (C, \gamma) \approx^{\mathfrak{R}_2} (N_{BC}, \gamma)$. где $N_{BC} = (B \,{}_{\overline{\beta}}\|_{\gamma}\, C)$.
Покажем, что для $N_{AC} = (A \,{}_{\overline{\alpha}}\|_{\gamma}\, C)$
$\mathfrak{R}_3 = \{(M_C, M_C + M_A) \mid \exists M_B \in [M_{0B}\rangle : (M_B, M_B + M_A) \in \mathfrak{R}_1, (M_C, M_C + M_B) \in \mathfrak{R}_2\}$ отношение бисимуляции $(C, \gamma) \approx^{\mathfrak{R}_3} (N_{AC}, \gamma)$:

1. $(M_{0C}, M_{0AC} = M_{0A} + M_{0C}) \in \mathfrak{R}_3$

2. Если $(M_{1C}, M_{1A} + M_{1C}) \in \mathfrak{R}_3$ и $W \in (\mathcal{M}(Vis))^*$, то
из построения $\mathfrak{R}_3$: $\exists M_{1B} : (M_{1B}, M_{1B} + M_{1A}) \in \mathfrak{R}_1, (M_{1C}, M_{1C} + M_{1B}) \in \mathfrak{R}_2$,

(a) если $(M_{1A} + M_{1C})\ [\Theta_{AC} = \Theta_A + \Theta_C\rangle_\gamma\ (M_{2A} + M_{2C})$, $\sigma_\gamma(\Theta_{AC}) = W$, то из определения N_{AB}: $\sigma_\gamma(\Theta_C) = \overline{\sigma_{\overline{\alpha}}(\Theta_\alpha)} = W$ и $M_{1A}(W\rangle_\alpha M_{2A}$ и $M_{1C}(W\rangle_\gamma M_{2C}$,
из отношения *допускает* $B \,^{\beta} \succ^{\gamma} C : (M_{1C}, M_{1C} + M_{1B}) \in \mathfrak{R}_2$ и $M_{1C}(W\rangle_\gamma M_{2C}$
$\Longrightarrow (M_{1C} + M_{1B})(W\rangle_\gamma (M_{2C} + M_{2B}) \Longrightarrow (M_{2C}, M_{2C} + M_{2B}) \in \mathfrak{R}_2$
из определения N_{BC}: $(M_{1C} + M_{1B})(W\rangle_\gamma (M_{2C} + M_{2B}) \Longrightarrow M_{1B}(W\rangle_\beta M_{2B}$,
учитывая $M_{1A}(W\rangle_\alpha M_{2A} \Longrightarrow (M_{2B}, M_{2B} + M_{2A}) \in \mathfrak{R}_1 \Longrightarrow (M_{2C}, M_{2A} + M_{2AC}) \in \mathfrak{R}_3$

(b) если $M_{1C}[\Theta_{AC}\rangle_\gamma M_{2C}$, $\sigma_\gamma(\Theta_C) = W$, то
из отношения *допускает* $B^b \succ^c C$ и $A^\beta \succ^\gamma B : M_{1B}(W\rangle_\beta M_{2B}$ и $M_{1A}(W\rangle_\beta M_{2A}$
$\Longrightarrow (M_{2C}, M_{2C} + M_{2B}) \in \mathfrak{R}_2$, $(M_{2B}, M_{2A} + M_{2B}) \in \mathfrak{R}_1 \Longrightarrow (M_{2C}, M_{2A} + M_{2C})) \in \mathfrak{R}_3$; ■

Утверждение 11. *Об антисимметричности отношения "допускает"*

Пусть A и B - простые сети Петри с пометками $\alpha = \langle \Delta_a, \sigma_a \rangle$ и $\beta = \langle \Delta_b, \sigma_b \rangle$ соответственно. Тогда если $(A, \alpha) \succ (B, \beta)$ и $(B, \beta) \succ (A, \alpha) \Longrightarrow (A, \alpha) \approx^{\mathfrak{R}} (B, \beta)$ ■ 11

Док-во: Пусть $N_{AB} = (A \,{}_{\overline{\alpha}}\|_{\beta}\, B)$ – сеть Петри, полученная в результате синхронизации сетей A и B по пометкам $\overline{\alpha}$ и β.

Тогда обозначим через $\mathfrak{R}_1$ и $\mathfrak{R}_2$ отношения *допускает* из начальных условий: $\mathfrak{R}_1 = \{(M_B, M_B + M_B)\} \subseteq [M_{0B}\rangle \times [M_{0AB}\rangle : (B, \beta) \approx^{\mathfrak{R}_1} (N_{AB}, \beta)$,
$\mathfrak{R}_2 = \{(M_A, M_B + M_A)\} \subseteq [M_{0A}\rangle \times [M_{0AB}\rangle : (A, \alpha) \approx^{\mathfrak{R}_2} (N_{AB}, \overline{\overline{\alpha}})$.
Тогда $\mathfrak{R} = \{(M_A, M_B) | (M_A, M_A + M_B) \in \mathfrak{R}_1$ или $(M_B, M_A + M_B) \in \mathfrak{R}_2\}$ отношение бисимуляции $(A, \alpha) \approx^{\mathfrak{R}} (B, \beta)$ ■

4. Метод верификации

В настоящей статье предлагается метод раздельной верификации (проверки работоспособности) классов и программ, использующих классы. Основные технологические этапы применения этого метода можно сформулировать следующим образом:

1. *спецификация требований к классу в виде корректных последовательностей вызовов методов класса - протокола класса;*

2. *построение моделей реализации класса и программы, использующей класс;*

3. *проверка модели реализации класса на соответствие протоколу;*

4. *проверка последовательности вызовов методов класса в модели программы, использующей класс, на соответствие протоколу класса.*

В основе предлагаемого метода верификации лежит проверка двух сетей на соответствие друг другу. На неформальном уровне соответствие сетей друг другу воспринимается различным образом в зависимости от их предназначения. В предыдущем разделе было введено новое отношение *допускает*, которое предоставляет формальный способ описания соответствия между сетями Петри, путём сравнения их поведения. Для проверки отношения *допускает* на практике следует использовать граф состояний сети Петри.

В данной статье используется понятие графа состояний, определяемое следующим образом:

Определение 12. *Граф состояний простой сети Петри*

Граф состояний простой сети Петри Σ определяется как набор $G = \langle X, U \rangle$, где

1. X - множество вершин графа, $X = [M_0\rangle = \{M' \mid M_0[\rangle M'\}$, совпадающее с множеством всех достижимых маркировок,

2. $U \subseteq X \times X \times \mu T$ есть множество рёбер графа, причём $u = \langle M_1, M_2, \Theta \rangle \in U \Longrightarrow M_1[\Theta\rangle M_2$.

■ 12

Для удобства обращения с графом состояний его рёбра помечаются множеством переходов, или шагов, посредством которых сеть может перейти из состояния, определяемого начальной вершиной ребра, в состояние, определяемое конечной вершиной ребра. Граф состояний сети Петри представляет собой конечный автомат, эквивалентный исходной сети Петри. Граф состояний сети Петри является основным инструментом обнаружения дедлоков – таких состояний, при которых не может сработать ни один переход.

Утверждение 9 определяет алгоритм проверки отношения *допускает* между сетями Петри A и B. Для этого необходимо построить граф состояний композиции сетей $N_{AB} = (A \,{}_a\|_b\, B)$ и проверить, что в каждом узле графа состояний участвуют все варианты срабатывания шагов из сети B[5].

В отношении между программой и классом протокол класса описывает все возможные корректные способы использования класса. Поэтому корректно использующая класс программа должна вести себя способом, предусмотренным протоколом. Т.е. поведение программы "уже", чем поведение протокола. В отношении между реализацией класса и протоколом протокол класса описывает только те варианты поведения класса, которые должны корректно обрабатываться реализацией. Реализация класса может вести себя ошибочно, если использовать варианты, не предусмотренные в протоколе. Т.е. протокол сужает варианты использования класса. **Используя отношение *допускает* , можно утверждать, что в программе корректно используются классы, если модели реализации всех классов, использованных в программе, *допускают* их протоколы, а протоколы классов *допускают* использование классов в модели программы.** При этом под моделью программы подразумевается модель основной функции программы, а под моделью реализации класса рассматривается объединение моделей методов этого класса. В таком случае, благодаря утверждению 10, будет справедливо, что реализации классов *допускают* программу. То есть поведение сети Петри, моделирующей программу, полностью вкладывается в корректное поведение классов.

Таким образом, предлагаемый метод верификации состоит в том, что и разработчику класса, и программисту – пользователю класса достаточно доказать совместимость их реализации с протоколом класса. Проверка соответствия реализации класса протоколу может быть более сложной, чем проверка единичной программы, однако это усложнение компенсируется, во-первых, более простой проверкой использования класса в программе, а во-вторых, одноразовостью проверки корректности класса для всех программ, использующих его.

[5]Это следует из третьего пункта утверждения 9, согласно которому если сеть A допускает B, то существует такое отношение бисимуляции, в котором каждой маркировке M_b сети B ставится в соответствие маркировка вида $M_a + M_b$ сети N_{AB}. Так как в графе состояния сети N_{AB} перечислены все маркировки этой сети, остаётся проверить, что часть "M_b" маркировок предусматривает все срабатывания сети B.

5. Пример применения метода верификации

5.1. Пример C++ класса. Построение протокола класса

Возьмём в качестве примера класс IndexList, реализующий функциональность индексированного списка. Пусть в качестве индекса используется строка символов и на каждый индекс может существовать не более одной записи в списке. Функциональность класса обеспечивается реализацией функций добавления, удаления и поиска элементов в списке. Декларация такого класса могла бы выглядеть следующим образом:

```
class IndexList{ //Декларация класса IndexList
public:
    IndexList(); // конструктор класса, инициализирующий внутр. переменные
    ~IndexList(); // деструктор класса, освобождающий выделенную память
private:
    :           //внутренние функции класса
public:
    bool Add(const char *index,void *pElement); // добавление
                                                // элемента в список
    bool Del(int pos); // удаление элемента из списка
    bool MakeIndex(); // построение индекса из списка
    bool MakeList(); // переход к редактированию списка
    int Find(const char *index); // поиск позиции по индексу
    void* Get(int pos); // получение элемента по позиции
    const char *Index(int pos); // получение индекса по позиции
    int Count(); // получить количество элементов индексированного списка
};
```

С классом IndexList имеют дело два типа программистов – разработчики и пользователи. При этом разработчик может выбрать один из множества вариантов хранения данных: в виде одномерного или многомерного, буферизированного или небуферизированного массива, в виде дерева или в виде списка. Кроме того, могут использоваться стратегии немедленного, отложенного или частичного индексирования списка. В зависимости от конкретной реализации класса его применение будет эффективным в различных ситуациях. От разработчика пользователю информация о классе IndexList обычно передаётся в виде декларации класса и сопровождающей документации в виде краткого описания методов класса. При этом описание особенностей реализации класса передаётся в неформальной форме.

Рассмотрим реализацию этого класса, имеющую по замыслу разработчика два логических состояния. В первом состоянии происходит добавление или удаление элементов из списка. Во втором состоянии список используется для быстрого поиска элементов по индексу. Добавление элементов с сохранением индексированности списка является процедурой достаточно ресурсозатратной, так как добавление единственного элемента может потребовать почти полного перестроения индекса. Поэтому в первом состоянии список элементов является неиндексированным и не

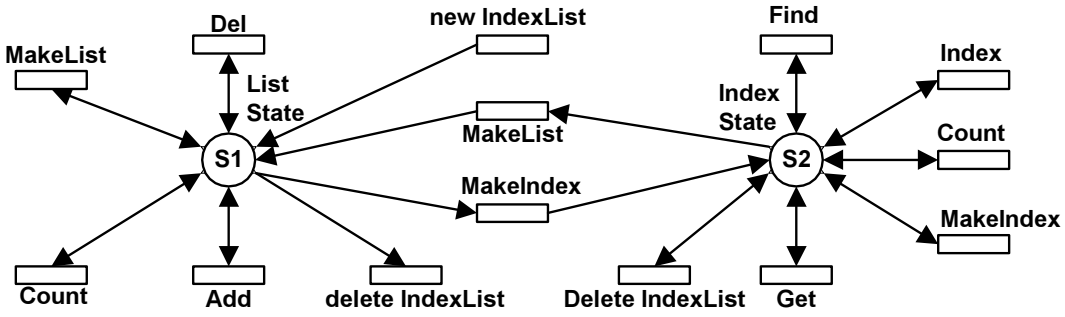

Рис. 2. Простая Сеть Петри Np, описывающая протокол класса IndexList

может быть использован для поиска по индексу, то есть поиск элементов в первом состоянии запрещён. Во втором состоянии класса список является индексированным, но добавление элементов в список приводит индекс в невалидное состояние и поэтому запрещено. По замыслу разработчика для эффективного использования класса работа с ним должна состоять из длинных серий добавлений или удалений объектов и длинных серий поисков объектов по индексу. В современных объектно-ориентированных языках программирования нет средств, способных обеспечить подобный сценарий работы, поэтому разработчиком класса предусмотрены выделенные функции перехода из состояния редактирования к индексированному состоянию и обратно, а также аварийное завершение работы (авост) при попытках вызова запрещённых в определённом состоянии функций. Такая реализация класса заставляет программиста задумываться над каждым обращением к методам класса и тем самым следовать корректному сценарию. Однако использование класса при этом становится небезопасным и может привести к нежелательным авостам.

Этот класс выбран в качестве примера, так как, во-первых, корректное завершение его функций не зависит от параметров, с которыми были вызваны функции, а во-вторых, зависит от состояния переменных – членов класса, имеющих малое количество значений. В общем случае про такие функции принято говорить, что они обладают побочным эффектом. Анализ программ, использующих функции с побочными эффектами, сильно затруднён по сравнению с анализом программ, использующих только функции без побочных эффектов, так как все побочные переменные, используемые функциями с побочными эффектами, надо рассматривать как дополнительные параметры вызова функции, не указанные во время вызова. Побочные переменные могут изменяться в любой части программы, вследствие этого исчезает возможность проверки программы по частям. Однако при правильной разработке класса побочные переменные используются для уменьшения количества параметров только методов класса, и таким образом происходит локализация побочных эффектов внутри реализации класса.

На рисунке 2 представлена простая сеть Петри Np, составленная в соответствии с замыслом разработчика, она описывает все допустимые последовательности вызова функций класса IndexList, поэтому будем называть её протоколом класса. Место S1 сети Np изображает состояние, в котором индексированный список может редактироваться. Место S2 – состояние использования индекса класса. Переход из одного состояния в другое происходит через события вызова методов $MakeList$ и

$MakeIndex$. Чтобы не загромождать рисунок, для переходов, связанных с местами дугами, ведущими в обоих направлениях, дуги объединены в одну двунаправленную. В начальном состоянии сеть Np не содержит токенов – что означает отсутствие экземпляров класса. Токены в сети добавляются каждый раз при вызове конструктора класса и удаляются из сети при вызове деструктора. Сеть Петри Np точно описывает замысел разработчика по использованию класса.

5.2. Пример программы. Построение модели программы

Первым этапом предлагаемого метода верификации является разработка протокола класса. Так же, как и в случае сетевых протоколов, – это работа, предшествующая реализации класса. Протокол класса описывает замысел разработчика и, в частности, может использоваться как техническое задание для дальнейшего программирования, а также как формальная техническая документация при передаче класса в эксплуатацию программистам-пользователям. В разделе 5.1. приведено описание протокола класса IndexList, подходящее для предлагаемого метода верификации.

Проверка корректности реализации класса IndexList является технически сложной, но реализуемой задачей. Предположим, что корректная реализация класса получена и проверена на соответствие протоколу, и рассмотрим вторую часть раздельной проверки работоспособности программ сформулированного ранее метода верификации.

В качестве примера применения класса IndexList рассмотрим задачу слияния двух словарей. Алгоритм задачи состоит из следующих шагов. Сначала из двух файлов считываются индексированные списки словарей иностранных слов. Далее каждое слово из второго словаря проверяется на наличие в первом. Если слово не присутствует в первом словаре, то оно в него добавляется. Ниже приведён пример текста программы на языке C++, который использует описанные сторонним программистом классы AFile и IndexList. Программа состоит из двух функций, реализующих чтение и запись индексированного списка из файла и в файл, а также основной функции main, которая реализует алгоритм слияния индексированных списков словарей:

```
#include "AFile.h"      // Программа слияния двух словарей на языке C++
#include "IndexList.h"

int ReadFromFile(const char *filename,IndexList &lst)
{
    AFile f1(filename,AFile::ReadMode);
    const char *index,const char* translation;
    int count=0;
    lst.MakeList();
    while(!f1.EOF())
    {
        f1.Read2String(index,translation);
        lst.Add(index,translation);
```

```
        count++;
    }
    lst.MakeIndex();
    return count;
}
//-------------------------------------
int WriteToFile(const char *filename,IndexList &lst) {
    AFile f1(filename,AFile::WriteMode);
    const char *index,const char* translation;
    int count=lst.Count();
    int i;
    lst.MakeIndex();
    for (int i=0;i<count;i++)
    {
        f1.Write2String(lst.Index(pos),(const char*)lst.Get(pos));
    }
    return count;
}
//-------------------------------------
int main()
{
    IndexList lst1,lst2;
    ReadFromFile("FileName1",lst1);
    int count=ReadFromFile("FileName2",lst2);
    const char *index;
    for (int i=0;i<count;i++)
    {
        index=lst2.Index(pos);
        if (lst1.Find(index)<0)
            lst1.Add(index,lst2.Get(pos));
    }
    WriteToFile("FileName1",lst1);
    return true;
}
```

Язык C++ предоставляет программисту фиксированный набор стандартных императивных семантических конструкций типа for, if, case, while и расширяемые возможности за счёт функций и классов. Основой формирования потока управления программой является вызов функции – вся программа является вызовом функции main, а функция main состоит из вызовов других функций. В терминах сетей Петри вызов функции в программе может быть промоделирован в виде последовательности из перехода, места и перехода. Переходы моделируют начало и конец вызова функции, по правилам прочтения сетей Петри – мгновенные неделимые действия, а место – состояние выполнения функции, в котором система может пребывать произвольное, неопределённое количество времени. При наличии модели исполне-

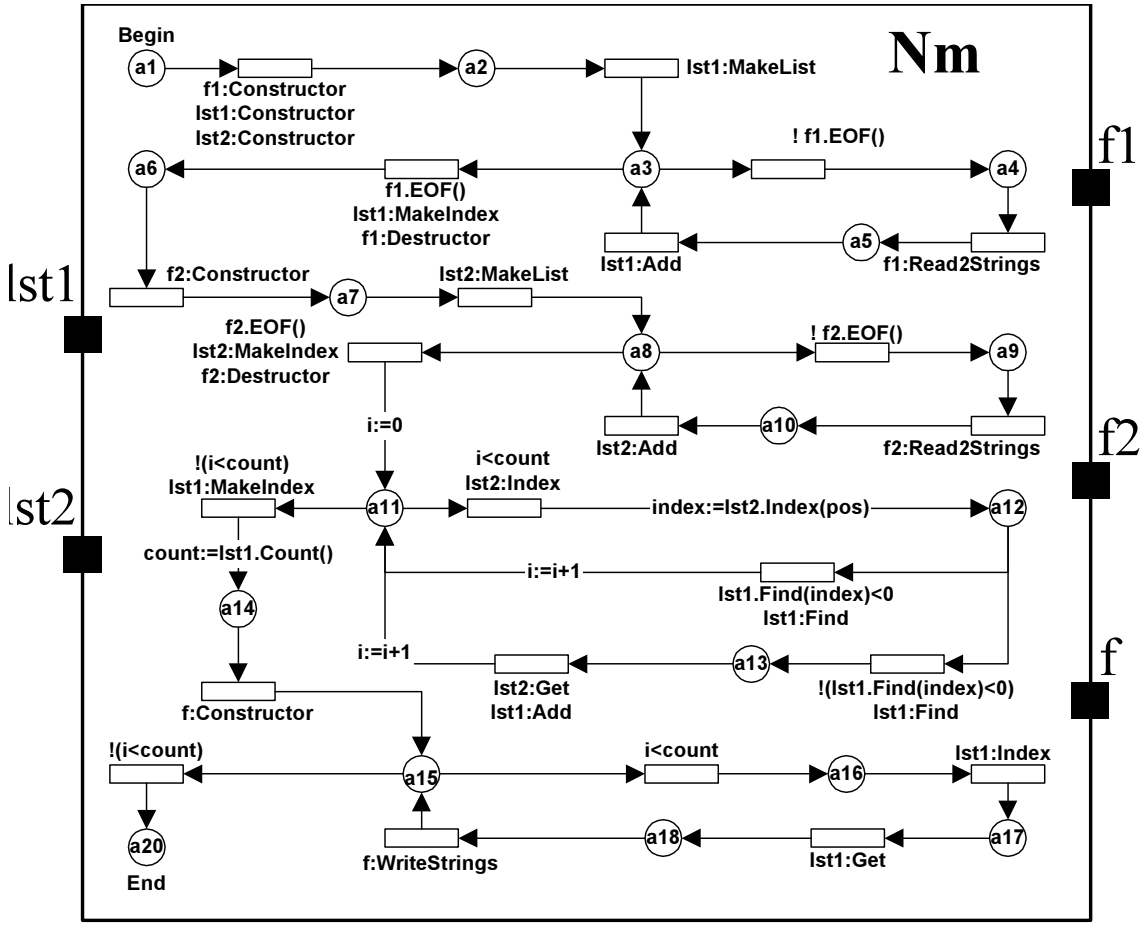

Рис. 3. Простая сеть Петри Nm, моделирующая программу слияния двух словарей и точки доступа к модели программы

ния функции в терминах сетей Петри вызов функции может быть синхронизован с её исполнением или же просто заменён на модель исполнения функции. В некоторых случаях для моделирования вызова функций вместо предложенной последовательности мест и переходов можно использовать один-единственный переход – это означает, что выполнение функции занимает незначимое для анализа время, и нахождением системы в состоянии выполнения этой функции можно пренебречь.

В протоколе класса IndexList каждый метод класса представлен одним или несколькими одиночными переходами, моделирующими вызов метода класса в различных состояних класса. Потому в модели использования класса вызовы методов этого класса также необходимо изображать одиночными переходами. Построенная для программы слияния словарей с учётом вышеизложенных замечаний[6] модель использования классов изображена на рисунке 3. Эта модель визуально разделяется на четыре части. Первые две части моделируют считывание двух словарей из различных файлов, третья часть моделирует слияние словарей, а четвёртая –

[6]Модель программы может быть построена автоматическим способом, однако так как метод и алгоритм построения модели не являются предметом рассмотрения в данной статье, будем считать, что модель программы строится "вручную".

запись объединённого словаря в файл. За исключением третьей части все остальные являются вставленными непосредственно в месте вызова моделями исполнения соответствующих функций, построенными из исходного текста программы. Модель программы разработана в терминах простых сетей Петри, но для увеличения информативности модели на некоторых переходах и дугах нанесены соответствующие цветным сетям Петри операции над данными – предикаты и выражения. Кроме того, переходы помечены различными пометками образца: <наименование точки доступа> : <символы пометки>.

Для проверки соответствия последовательности вызовов методов классов своим протоколам на модели программы все вызовы отображены как пометка переходов, осуществляющих вызов, точками доступа с наименованиями, совпадающими с наименованиями переменных. На рисунке 3 показано, что в сети Nm, моделирующей программу, имеется пять точек доступа по переходам: f1, f2, f к протоколам класса AFile и lst1, lst2 к протоколам класса IndexList.

5.3. Проверка работоспособности программы

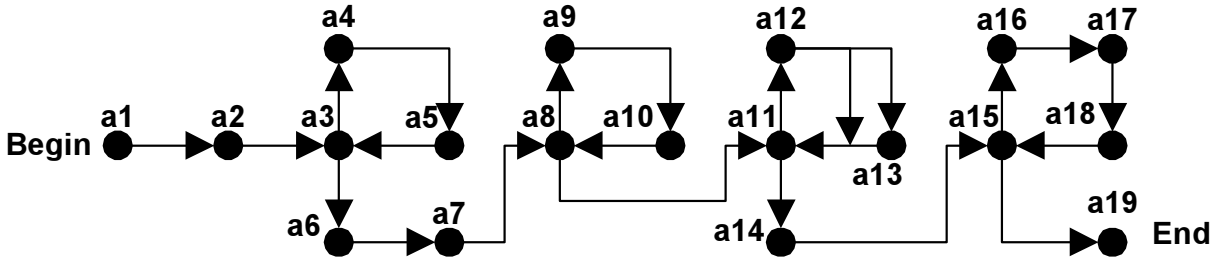

Рис. 4. Граф состояния модели программы

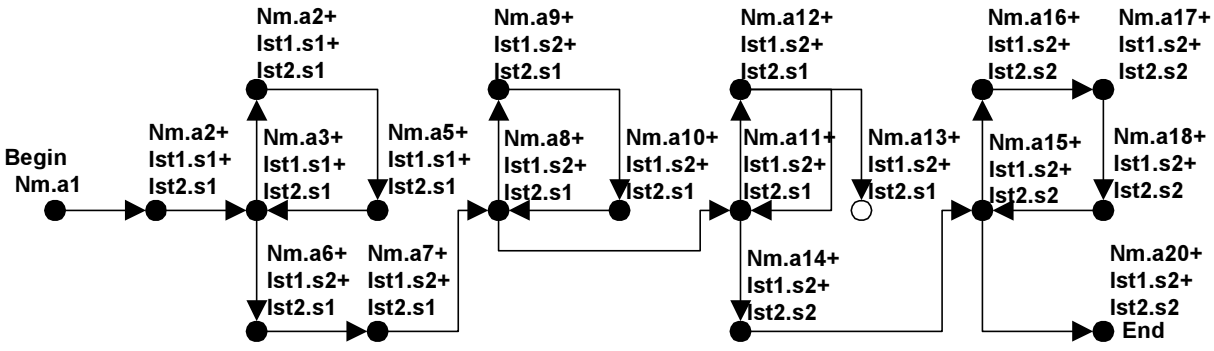

Рис. 5. Граф состояния модели программы, синхронизованной с протоколами класса IndexList

Для проверки работоспособности программ слияния словарей построено два графа состояния, изображённых на рисунках 4 и 5. Первый граф состояния, на рисунке 4, соответствует конечному автомату алгоритма слияния словарей, он построен

без синхронизации с классами. Второй граф состояния приведён для модели программы, синхронизованной с протоколами классов переменных lst1 и lst2, причём состояния на этом графе помечены как суперпозиция состояний всех сетей с использованием синтаксиса: <наименование сети>.<состояние сети>. На втором графе отмечен дедлок в состоянии Nm.a13+lst1.s2+lst2.s2, этот дедлок говорит о том, что обращение к переменной lst1 из состояния a13 приводит к дедлоку. Если посмотреть исходный текст программы, то это означает, что вызов метода Add в данный момент времени приведёт к авосту, так как экземпляр класса lst1 в этом месте находится в состоянии использования индексирования списка, и добавление в список запрещено. Таким образом, проверка программы завершена и обнаружена ошибка. Далее всё будет зависеть от того, какое решение примет программист. Он может либо добавить вызовы методов перехода из одного состояния класса IndexList в другое, либо исправить алгоритм более кардинальным образом. Например, добавление в индексированный список lst1 может быть отложено до полного сравнения словарей, а пока происходит сравнение, не найденные в списке lst1 элементы можно сохранять в промежуточный список.

6. Заключение

Формальная верификация – доказательство с помощью формальных методов правильности или неправильности программы в соответствии с формальным описанием её свойств. Методы верификации программ можно условно разделить на две категории. К первой категории относятся методы, подобные доказательству теорем (Theorem Proving), где программа в целом рассматривается как формальное высказывание, для которого необходимо доказать те или иные предикаты свойств. Основу таким методам предоставляет аксиоматическая логика Хоара, предписывающая операторам языка программирования те или иные аксиомы [10, 6, 7]. Ко второй категории относятся методы верификации на моделях (Model Checking) [14, 15]. Суть этих методов описывается в трёх стадиях. На первой стадии строится модель программы, обычно основанная на системах переходов. На второй стадии модель программы дополняется спецификацией свойств-требований, которыми должна обладать программа. Обычно спецификации задаются на языке формальной логики, как правило, применяются динамические логики, временные логики и их варианты с неподвижными точками. На третьей стадии непосредственно производится верификация свойств программы. В процессе работы алгоритма глобальной проверки на модели строится множество состояний модели, в которых выполняется спецификация, и алгоритм локальной проверки на модели завершает работу с ответом true, если на модели не найдено состояний, не удовлетворяющих спецификации, либо строит ошибочную трассу, показывающую, почему не выполняется спецификация.

Объектно-ориентированная методология внесла новые стандарты в программирование [9] . Преимущество повторного использования компонент программ оказалось столь велико, что и императивные, и функциональные языки вынуждены перестраиваться для её применения. Основой верификации объектно-ориентированных программ де-факто является концепция Design By Contract [12, 13], предполагаю-

щая описание специальных утверждений-контрактов, которые должны принимать истинные значения в соответствующих местах программы. Существуют различные типы контрактов: элементарные утверждения (asserts), инварианты классов (class invariants), предусловия (function entry pre-conditions) и постусловия (function exit postconditions) для функций. Сочетание концепции DesignByContract и Model Checking или Theorem proving методов верификации позволяет проводить анализ корректности объектно-ориентированных программ.

Предложенный в настоящей статье метод верификации объектно-ориентированных программ относится к Model Checking категории, в несколько видоизменённом виде реализующий концепцию Design By Contract. Для моделирования программы используются помеченные сети Петри, а в качестве спецификации требований выступают сети Петри, описывающие корректное поведение классов. Учитывая гораздо меньшие выразительные возможности помеченных сетей Петри по сравнению с языками формальной логики, такой подход существенно сужает возможности описания свойств класса, одновременно максимально упрощая процесс верификации.

Для ряда широко применяемых классов, реализующих функциональность файлов, сокетов, классов доступа к базам данных, классов оконного интерфейса, описание протокола класса в виде корректных последовательностей вызова методов класса может упростить задачу поиска некорректностей в программах, использующих эти классы. Однако область применимости предложенного подхода трудно ограничить без практической его реализации, над которой работает в настоящее время автор статьи. В случае его успешной реализации автор рассчитывает в дальнейшем обобщить предлагаемый подход, во-первых, на классы, обладающие виртуальным наследованием, а во-вторых, на более мощный моделирующий аппарат цветных сетей Петри.

Список литературы

1. *Анисимов Н.А.* Формальная модель для разработки и формального описания протоколов // Автоматика и вычислительная техника. – 1988. – № 6. – С. 3 – 10.

2. *Анисимов Н.А., Голенков Е.А., Харитонов Д.И.* Композициональный подход к разработке параллельных и распределенных систем на основе сетей Петри // Программирование. – 2001. – № 6. – С. 30 – 43.

3. *Бандман О.Л.* Проверка корректности сетевых протоколов с помощью сетей Петри // Автоматика и вычислительная техника. – 1986. – № 6. – С. 82 – 91.

4. *Непомнящий В.А., Рякин О.М.* Прикладные методы верификации программ // М.: Радио и связь, 1988.

5. *Лисица А.П., Немытых А.П.* Верификация как параметризованное тестирование (эксперименты с суперкомпилятором SCP4) // Программирование. – 2007. – № 1. – С. 22 – 34.

6. *Непомнящий В.А., Анореев И.С., Михайлов И.Н., Промский А.В.* На пути к верификации С-программ. Язык С-light // Тезисы докладов. Конференция, посвященная 90-летию со дня рождения Алексея Андреевича Ляпунова. – Новосибирск, 2001.

7. *Непомнящий В.А., Анореев И.С., Дубрановский И.В., Промский А.В.* На пути к верификации С#-программ: Трёхуровневый подход // Программирование. – 2006. – № 4. – С. 4 – 20.

8. *Липаев В.В.* Методология верификации и тестирования крупномасштабных программных средств // Программирование. – 2003. – № 4. – С. 7 – 24.

9. *Мейер Б.* Объектно-ориентированное конструирование программных систем. – М.: Русская редакция, 2005.

10. *Hoare C.A.R.* An Axiomatic Basis for Computer Programming // Comm. ACM. – 1969. – V.12, N 10. – P. 576 – 580. The Queen's University of Belfast,* Northern Ireland

11. *Petri K.A.* Kommunication mit Automaten, Schriften des Rheinish // Westfalischen Institutes fur Instrumentelle Mathematik and der Universitat. – Bonn, 1962.

12. *Müller P.* Modular Specification and Verification of Object-Oriented Programs. – Springer-Verlag, 2001.

13. *Barnett M., DeLine R., Fähndrich M., Rustan K., Leino M., Schulte W.* Verification of Object-Oriented Programs with Invariants // Journal of Object Technology. – 2004. – 3. – P. 27 – 56.

14. *Quielle J., Sifakis J.* Specification and verification of concurrent systems in CESAR // Proceedings 5-th International Symposium on Programming, LNCS 137. – 1982. – P. 337 – 351.

15. *Clarke E.M., Emerson E.A.* Synthesis of synchronisation skeletons for branching time logic // Logic of Programs, LNCS 131. – 1981. – P. 52 – 71.

Separable object-oriented program verification with C++ class protocol definition in terms of Petri nets

Kharitonov D.I.

Keywords: programming languages, formal program verification, object-oriented programming, Petri nets

Formal proving of program correctness is the only reliable method that ensures proper functioning of program systems. Existing verification methods are quite complicated in practice, that obstructs their extensive application. In this article an approach using "Design By Contract" conception is proposed, where class protocols formulated in terms of Petri nets makes it possible to analyze workability of classes and programs separately that facilitates the procedure of program correctness checking.

Сведения об авторах: Харитонов Дмитрий Иванович,
канд. техн. наук, науч. сотр. Института автоматики и процессов управления Дальневосточного отделения РАН