СИСТЕМЫ И СРЕДСТВА ИНФОРМАТИКИ, 2018. Т. 28. № 2. С. 99–115

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МЕТОДА ГЕНЕРАЦИИ СЕТЕЙ ПЕТРИ БОЛЬШОГО РАЗМЕРА*

Д. И. Харитонов[1], *Г. В. Тарасов*[2], *Д. В. Леонтьев*[3]

Аннотация: Описана программная реализация метода генерации сетей Петри (СП) для числа мест и переходов свыше одного миллиона. Особое внимание уделяется внутренним структурам данных и алгоритмической сложности поставленной задачи. Представленные результаты позволили получить программное средство с вычислительной сложностью $O(n)$, где n — число элементов модели. Описанные теоретические исследования подтверждаются практическими результатами измерения производительности.

Ключевые слова: сети Петри; модель программы; структуры данных; анализ алгоритмов; объектно-ориентированное программирование

DOI: 10.14357/08696527180208

1 Введение

Известно, что одной из главных проблем верификации методом проверки модели является взрыв числа состояний. Эта проблема не может быть решена в принципе ввиду неограниченной сложности исследуемых систем. Однако разработан ряд методов, которые позволяют рассматривать модели с очень большим числом состояний. В частности, ведутся исследования в области модульного представления СП, их анализа по частям с использованием распределенных и параллельных технологий обработки [1–5]. Построение моделей большого размера, описывающих поведение реальных систем, невозможно выполнить вручную. Разработано множество различных инструментальных средств для построения и анализа моделей аппаратных систем, распределенных и параллельных программных систем [6], в том числе с использованием различных расширений теории СП [7–9]. Начиная с 2011 г. в рамках ведущей международной ежегодной конференции по СП (International Conference on Application and Theory of Petri Nets and Concurrency) проводится конкурс программных

*Работа выполнена при поддержке Программы Президиума РАН (проект 0262-2014-0157) и Программы «Дальний Восток» (проект 15-I-4-025).

[1]Институт автоматики и процессов управления Дальневосточного отделения Российской академии наук, demiurg@dvo.ru

[2]Институт автоматики и процессов управления Дальневосточного отделения Российской академии наук, george@dvo.ru

[3]Институт автоматики и процессов управления Дальневосточного отделения Российской академии наук, devozh@dvo.ru

систем верификации моделей (Model Checking Contest @ Petri Nets). В 2016 г. для тестирования участникам конкурса были предложены модели, максимальное число элементов которых приблизилось к 400 000 (включая дуги).

Направление исследований, связанное с автоматической генерацией моделей больших размеров, интересно с практической точки зрения. Во-первых, их можно использовать в отладочных целях для тестирования производительности и эффективности существующих методов анализа и разработки новых методов. Именно эта цель преследуется организаторами упомянутого выше конкурса. Во-вторых, модели с большим числом элементов полезны в алгоритмических целях. Способы хранения СП в оперативной памяти вычислительной машины, алгоритмы обхода и поиска по дереву элементов модели, алгоритмы добавления, удаления и связывания элементов модели и т. п. превращаются в нетривиальные задачи, когда требуемый объем памяти для хранения элементов модели приближается к предельным возможностям современных вычислительных систем (десятки и сотни гигабайт в рамках одной рабочей станции или узла вычислительного кластера).

В данной работе описана программная реализация метода генерации СП большого размера (миллионы элементов и больше) с помощью шаблонов. Основное внимание уделяется описанию алгоритмов и структур данных, использованных при реализации программного средства. Данная работа является продолжением исследований алгоритма генерации больших моделей в терминах СП, предложенного авторами ранее [10].

Материал статьи представлен следующим образом. Вначале дается краткое описание основных определений понятия шаблона, операции слияния и алгоритма генерации большой СП. Далее подробно описываются структуры данных для хранения СП и шаблона в памяти вычислительной машины. В заключение проводится анализ сложности алгоритма и результаты тестирования его производительности для сетей различного размера.

2 Генерация моделей на основе шаблонов

В основе разработанного алгоритма генерации модели лежит понятие шаблона. Математически шаблон представляет собой СП, дополненную специальной пометкой, позволяющей связывать шаблоны между собой.

Определение 1. СП-шаблоном (или просто шаблоном) называется четверка элементов $X = (\Sigma, I, O, M_0)$, где

(1) $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$ — СП, называемая структурой шаблона, где S — множество мест; T — множество переходов; ${}^\bullet() \subseteq \mathcal{M}(S)$ — входная функция переходов (мультимножества инцидентных мест); $()^\bullet \subseteq \mathcal{M}(S)$ — выходная функция переходов;

(2) $I = \{\iota_1, \iota_2, \ldots, \iota_n\}$ — множество простых точек доступа (ТД), называемое входным интерфейсом;

(3) $O = \{\phi_1, \phi_2, \ldots, \phi_m\}$ — множество простых ТД, называемое выходным интерфейсом;

(4) $M_0 \in \mathcal{M}(S)$ — начальная маркировка шаблона.

Символом $\mathcal{M}(A)$ будем обозначать множество всех конечных мультимножеств на множестве A. Пустое мультимножество будем обозначать символом $\mathbf{0}$. Простой ТД будем называть пару элементов $\iota = \langle \mathrm{id}_i, \varrho_i \rangle$, где id_i — уникальный идентификатор ТД; $\varrho_i \in [S]$ — линейно упорядоченное подмножество мест, используемое ТД.

Два СП-шаблона могут быть слиты друг с другом при помощи операции слияния по простым ТД.

Определение 2. Пусть заданы два СП-шаблона $X_1 = (\Sigma_1, I_1, O_1, M_{01})$ и $X_2 = (\Sigma_2, I_2, O_2, M_{02})$ и две их простые ТД $\iota \in I_1$ и $\phi \in O_2$, где $\iota = \langle \mathrm{id}_1, \varrho_1 \rangle$, $\phi = \langle \mathrm{id}_2, \varrho_2 \rangle$, и мощности подмножеств мест этих ТД равны $|\varrho_1| = |\varrho_2|$. Тогда операция слияния СП-шаблонов X_1 и X_2 формирует новый шаблон $X = (\Sigma, I, O, M_0)$ такой, что

$$X = (X_1 \oplus X_2)|_{\phi}^{\iota} .$$

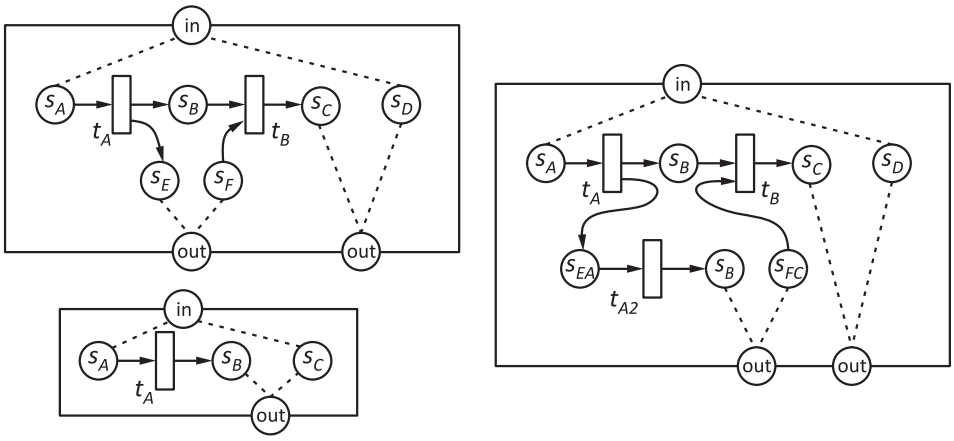

Рис. 1 Пример операции слияния шаблонов

Операцию слияния можно представить как формальное объединение двух шаблонов с последующим слиянием заданного ТД подмножества мест шаблонов. Формальное объединение структур шаблонов обозначено символом $\oplus$. Слияние по ТД обозначено вертикальной чертой с указанием ТД, участвующих в операции. На рис. 1 показан пример операции слияния.

Шаблоны могут быть объединены в коллекции.

Определение 3. Множество $\Pi = \{X_i = (\Sigma_i, I_i, O_i, M_{0i})\}$ называется коллекцией СП-шаблонов, если

(1) в ней определен стартовый шаблон X_0 такой, что $M_{00} \neq \mathbf{0}$, $|I_0| = 0$, $|O_0| > 0$;

(2) в ней определено несколько строительных шаблонов $|\{X_k \mid X_k \in \Pi, M_{0k} = \mathbf{0}, |I_k| = 1, |O_k| \geq 0\}| > 0$ (строительный шаблон называется завершающим, если его выходной интерфейс пуст: $|O_k| = 0$);

(3) все простые ТД во входных интерфейсах имеют пару в выходных интерфейсах, и наоборот:

$$\forall\, X_i \in \Pi, \quad \phi \in O_i \to \exists\, X_j \in \Pi, \quad \iota \in I_j : |\phi| = |\iota|\,;$$

$$\forall\, X_i \in \Pi, \quad \iota \in I_i \to \exists\, X_j \in \Pi, \quad \phi \in O_j : |\iota| = |\phi|\,.$$

Неформально под коллекцией можно понимать такую систему, в которой результат слияния стартового шаблона с любым другим строительным шаблоном дает новый стартовый шаблон. Развитие этого процесса можно представить следующим образом. Коллекция $\Pi_{n+1} = \{X_0^{n+1}, X_1, \ldots, X_k\}$ порождается коллекцией $\Pi_n = \{X_0^n, X_1, \ldots, X_k\}$, если $\exists\, \iota, \phi$ такие, что

$$X_0^{n+1} = (X_0^n \oplus X_j)\big|_{\phi}^{\iota}\,,$$

где $\iota \in I_j$ — входная ТД шаблона X_j, $\phi \in O_0$ — выходная ТД стартового шаблона. Результатом эволюции коллекции будем называть такой шаблон $X = \langle \Sigma, I, O, M_0 \rangle$, у которого $|I| = 0, |O| = 0$.

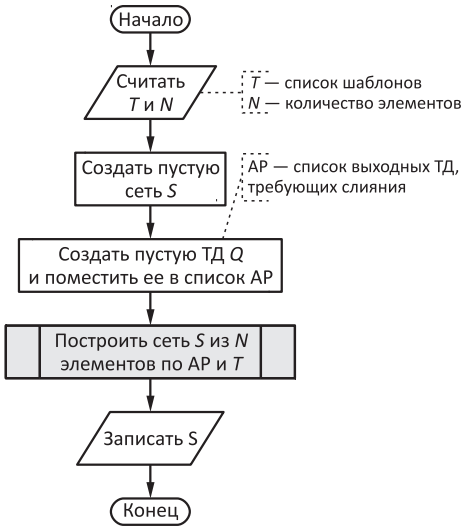

Рис. 2 Общий алгоритм построения сети

Опишем алгоритм, реализующий схему генерации СП с использованием заданного набора строительных шаблонов. Алгоритм состоит из пяти шагов, изображенных на рис. 2. На первом шаге из конфигурационного файла выполняется считывание коллекции шаблонов и параметров результирующей сети. Далее выполняется построение объекта пустой СП и списка выходных ТД для запуска итерационного алгоритма генерации сети. Список выходных ТД вначале содержит одну пустую стартовую ТД. После этого следует непосредственный запуск итерационной процедуры генерации. И на последнем шаге алгоритма результаты работы записываются в выходной файл.

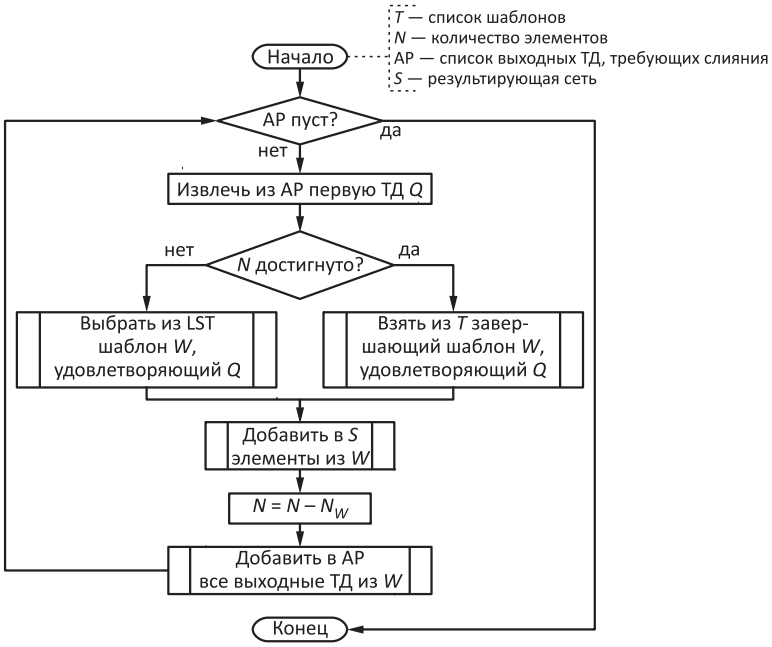

Рис. 3 Алгоритм итерационного заполнения сети

Основная работа по генерации большой СП выполняется итерационным алгоритмом, принимающим на вход список шаблонов, объект сети для генерации, число элементов для добавления в эту сеть, список шаблонов, а также список выходных ТД, по которым должна быть проведена операция слияния с каким-либо строительным шаблоном. Блок-схема алгоритма показана на рис. 3 и реализует эволюцию коллекции относительно результирующей сети S, которая служит стартовым шаблоном на каждом шаге итерации. Список AP содержит выходные ТД стартового шаблона. Когда на очередном шаге итерации оказывается, что все выходные ТД слиты с какими-либо строительными шаблонами, алгоритм заканчивает выполнение. Процедура выбора добавляемого шаблона может быть параметризована для получения результирующей сети с заданными характеристиками. Наибольшее значение с точки зрения времени выполнения имеет процедура добавления шаблона к результирующей сети, именно она определяет вычислительную сложность алгоритма относительно числа элементов.

Опишем, как можно эффективно реализовать предложенный алгоритм, а также опишем сопутствующие форматы и структуры данных, чтобы в результате получить программу генерации большой СП за $O(n)$. Информация будет представлена в следующем порядке. Сначала будет описано внутреннее представление СП в памяти ЭВМ, далее будет описан формат входных данных для алгоритма

генерации и затем будет описан формат представления шаблона в памяти ЭВМ. На каждом шаге описания будет уделяться особое внимание деталям, которые в совокупности дают эффективную и быструю реализацию программы генерации большой СП.

3 Представление сетей Петри в памяти вычислительной машины

Существуют два основных способа представления СП в памяти вычислительной машины. В первом варианте сеть может быть представлена матрицей инцидентности, а во втором — одним или несколькими списками своих элементов (мест, переходов и дуг). Оба подхода имеют свои преимущества и недостатки.

Основным преимуществом матричного представления является быстрый доступ к элементам сети и связям между ними. Вместе с тем использование матричного представления существенно увеличивает объем занимаемой памяти. Например, для СП, состоящей из 10^5 элементов, матрица инцидентности может занимать порядка 10 ГБ, что является условной границей возможностей современных персональных компьютеров. Для сокращения занимаемого пространства в памяти можно использовать разряженные матрицы, битовые маски и другие методы. Однако применение специальных методов хранения требует дополнительных средств и времени для выполнения операций добавления, удаления и поиска элементов в сети. Поскольку алгоритм генерации в значительной степени зависит от скорости выполнения операции добавления элемента в сеть, можно сказать, что матричный способ представления больших сетей будет неэффективен.

Представление на основе списков не обеспечивает быстрого доступа к элементам сети. Требуется применять дополнительные средства (например, использовать сортировку списков, составлять поисковые индексы) для реализации быстрого доступа к элементам сети. Однако представление СП в виде набора списков существенно упрощает операцию добавления элементов в сеть, а также позволяет разместить в памяти ЭВМ сети существенно большего размера без дополнительных расходов.

Для хранения СП в памяти ЭВМ был разработан формат, который содержит только значимую информацию об СП, достаточную для описания ее структурных и поведенческих свойств. Остальные данные будем считать незначимыми, полагая, что они могут быть воссозданы по требованию в процессе работы с сетью (например, индексы списков элементов для быстрого поиска). На рис. 4 представлена UML (unified modeling language) диаграмма классов, которая описывает способ представления СП в объектно-ориентированной программе. Показан только тот функционал, который важен для задачи генерации сети. Базовый класс PN3Item описывает общие свойства любого элемента СП, включая саму сеть. Каждый элемент содержит имя (атрибут name) и список предписаний (атрибут inscriptions). Предписания позволяют по мере необходимости наращивать описательные возможности СП. Например, работая только

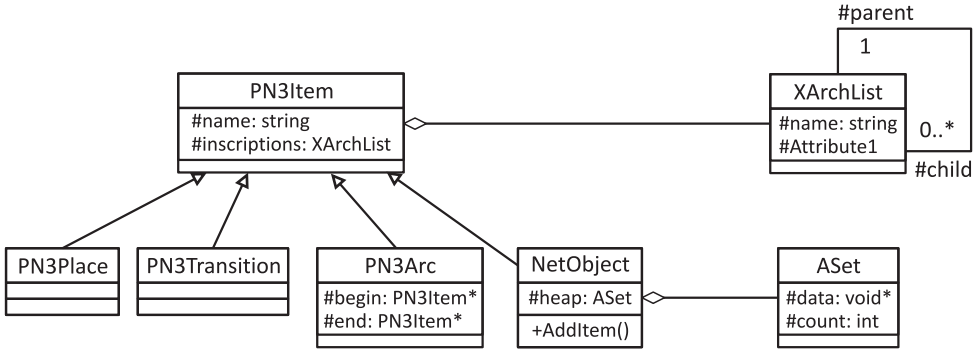

Рис. 4 Диаграмма классов для представления СП в памяти ЭВМ

с простыми СП, в предписаниях можно сохранять информацию о распределении фишек и кратности дуг. Работая с высокоуровневыми СП, в предписаниях можно сохранять информацию о типах данных, значениях, выражениях и других конструкциях языка описаний. Атрибут inscriptions представляет собой объект специального класса XArchList, реализующий структуру данных — дерево, в котором каждая вершина представляется парой строк: имя и значение. Классы PN3Place, PN3Transition и PN3Arc описывают поведение соответствующего элемента структуры СП. Класс NetObject описывает саму СП в виде множества элементов: мест, переходов и дуг. Хранение списка элементов сети выполняется отдельным классом ASet, представляющим собой структуру данных «линейный список».

Таким образом, для хранения одной СП должен быть создан экземпляр класса NetObject, в который добавляются объекты классов PN3Place, PN3Transition и PN3Arc. При необходимости вносятся предписания о кратности дуг и распределении фишек. Для метода AddItem, который добавляет некоторый элемент в сеть, при генерации можно сделать допущение об уникальности имени добавляемого элемента. Тогда AddItem можно реализовать на основе классического алгоритма добавления элемента в линейный список, стоимость которого составляет $O(1)$ [11].

4 Описание входных данных генерации большой сети

На вход алгоритму подается специальный конфигурационный файл, в котором дается описание всех шаблонов и параметры генерации сети. Опишем формат входных данных алгоритма генерации СП большого размера. Сначала представим общее описание синтаксиса конфигурационного файла. После этого определим синтаксис для записи шаблона и синтаксис записи параметров генерации. Далее представим простой пример генерации большой СП.

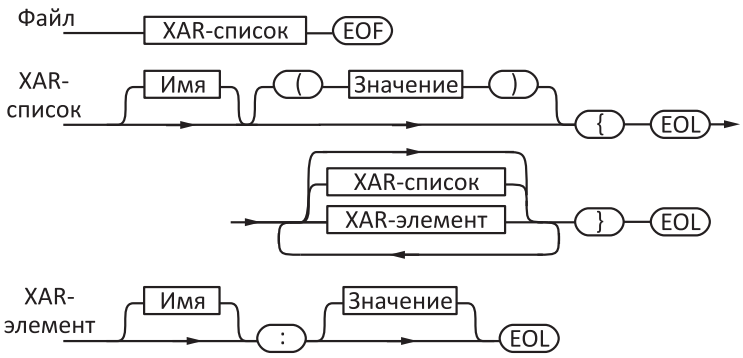

Рис. 5 Синтаксис XAR-файла

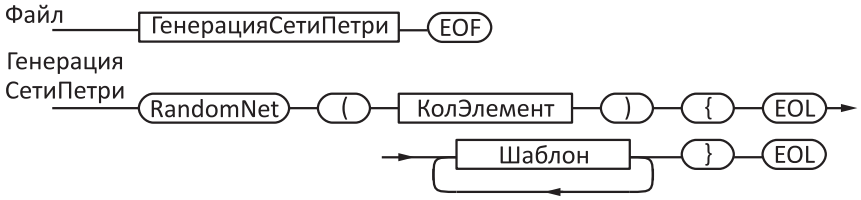

Рис. 6 Синтаксис конфигурационного файла генерации СП

За основу описания входных данных взят формат JSON (JavaScript object notation), реализованный в сильно упрощенной форме. Все описание представляет собой дерево объектов. Каждый объект состоит из двух строковых атрибутов: имя и значение. Каждый объект может иметь произвольное число дочерних объектов. Таким образом, описание конфигурации представляет собой дерево из пар строк. Далее данный формат будем называть XAR-форматом, а компоненты дерева — XAR-списком и XAR-элементом. Синтаксис содержимого файла в формате XAR показан в форме диаграмм Вирта на рис. 5.

На диаграмме нетерминалы обозначены прямоугольниками, терминалы — овалами. Терминал EOL обозначает символ конца строки, EOF — символ конца файла. Как видно из диаграммы, вся значимая информация в XAR-формате записывается в нетерминалах Имя и Значение. Распишем синтаксис записи настроек генерации СП, включающих описание шаблонов и параметров. Синтаксис конфигурационного файла показан на рис. 6.

Как видно, описание настроек начинается с описания служебного XAR-списка с именем RandomNet и значением количества элементов СП (мест и переходов), которые необходимо сгенерировать. Далее с новой строки следует описание шаблонов генерации в виде отдельных XAR-списков. Детальное описание нетерминала Шаблон показано на рис. 7.

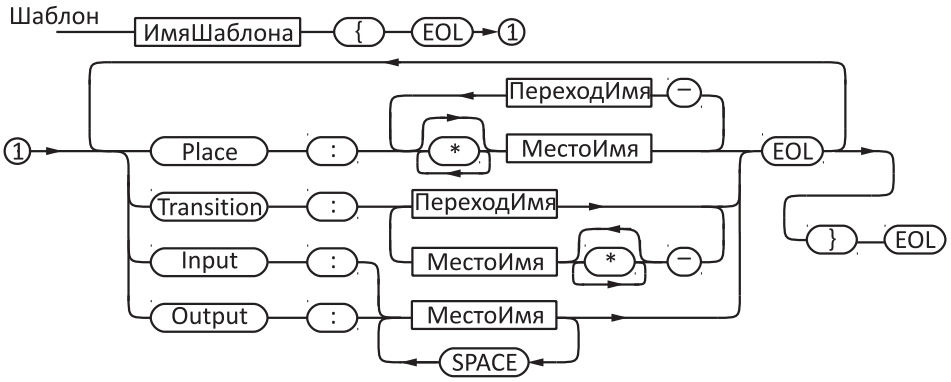

Рис. 7 Синтаксис описания шаблона СП

Текстовое представление шаблона включает описание графа СП, заданное терминалами Place и Transition, и описание ТД, по которым будет происходит слияние шаблонов. Суть использования каждой группы настроек рассмотрим на конкретном примере генерации большой СП, показанном на рис. 8. Вместо реального символа переноса строки (терминал EOL) для сокращения размера описания на рисунке будем использовать угловую стрелку. Для улучшения восприятия будем использовать отступы и пробельные символы. В реальном конфигурационном файле этот синтаксический «сахар» не требуется.

Результатом генерации с использованием конфигурационного файла, показанного на рис. 8, будет СП, состоящая из не менее 100 000 элементов (мест и переходов). Для генерации сети используется коллекция из четырех шаблонов. Шаблон BEGIN служит стартовым и имеет только одну исходящую ТД для слияния с другим шаблоном. Шаблон END — завершающий, он имеет

```
RandomNet(100000) {
BEGIN { ↲ Place:*s1 ↲ Place:s2 ↲ Output:s1 s2 ↲ }
END { ↲ Place:s1-t1-s2 ↲ Input:s1 s2 ↲ }
LINE {
     Place:s1-t1-s2 ↲ Place:s3
     Input:s1 s3 ↲ Output:s2 s3 ↲
}
BRANCH {
     Place:s1-t1-s2 ↲ Place:s1-t2-s3 ↲ Place:s4
     Input:s1 s4 ↲ Output:s2 s4 ↲ Output:s3 s4
}↲ }
```

Рис. 8 Пример файла генерации большой СП

только одну входящую ТД. Остальные два шаблона LINE и BRANCH являются структурными и позволяют увеличиваться сети либо в длину, либо в ширину.

Для описания структуры шаблона и его ТД разработана *линейная* форма записи элементов. Она позволяет описать граф СП в виде набора последовательностей имен элементов. Все соседние элементы в последовательности попарно связаны между собой дугами, следующими в одном направлении. Кратность дуги принимается равной единице. Имя элемента может неоднократно встречаться в рамках одной последовательности и в разных последовательностях. В первом случае такое появление равносильно появлению петли или цикла в графе. Во втором случае такое появление позволяет строить более сложные графы из набора простых линейных подграфов. Общее число дуг элемента в шаблоне, которыми он связан с другими элементами, равно количеству символов «тире» до и после имени данного элемента. Например, в шаблоне BRANCH место s1 имеет две исходящие дуги в переходы t1 и t2. Терминалы Place и Transition обозначают, какой из элементов СП будет первым в указанной последовательности. Учитывая, что по определению граф СП является двудольным, в последовательности нетрудно определить тип следующего элемента. Кратность дуг в данном варианте записи формируется за счет повторения пары связанных элементов. Для обозначения фишки в месте используется символ «звездочка» перед именем места. Последовательность из «звездочек» определяет число фишек в данном месте.

Описание ТД также задается набором последовательностей имен мест. Тип ТД определяется соответствующим терминалом, после чего следует перечисление через пробел имен мест СП, участвующих в операции слияния мест. Порядок перечисления мест имеет значение. Слияние двух шаблонов происходит по Input и Output ТД в том порядке, в котором они перечислены в описании этих ТД.

5 Представление шаблона в памяти вычислительной машины

Рассмотрим представление коллекции шаблонов в памяти ЭВМ. На рис. 9 показана UML-диаграмма классов, описывающая поведение шаблона и его входных и выходных ТД. Класс NBR_Prototype — это шаблон, класс NBR_Param — это входная или выходная ТД шаблона.

Шаблон в памяти ЭВМ хранится в виде набора списков и массивов, инициализация которых происходит в ходе считывания входных данных. Сначала заполняются списки имен элементов и расположения фишек. На UML-диаграмме в классе NBR_Prototype данные списки обозначены атрибутами Places, Transitions, Arcs и Tokens. Также формируются списки входных и выходных ТД (объекты класса NBR_Param), атрибуты Input и Output соответственно. Эта информация является текстовой и считывается из конфигурационного файла методом Load. После этого происходит подготовка шаблона к использованию в алгоритме генерации. Подготовка заключается в заполнении специальных целочисленных массивов, которые позволяют реализовать процедуру слияния шаблонов за $O(n)$, где n в данном случае — это число мест, участвующих

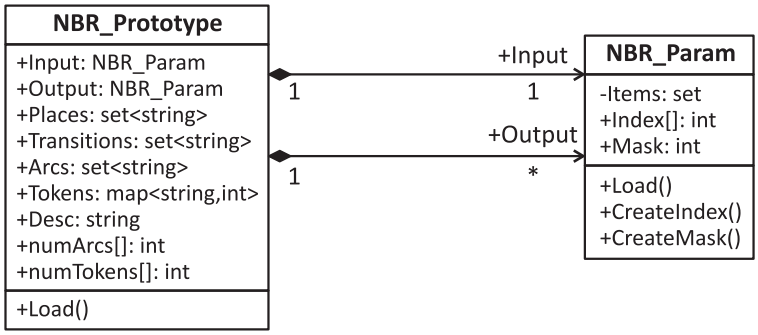

Рис. 9 Диаграмма классов, описывающая шаблон

в слиянии в одной ТД, и процедуру выбора шаблонов из коллекции за $O(1)$. Инициализируются следующие массивы: Index в NBR_Param хранит порядковый номер места в списке имен мест шаблона; numArcs и numTokens в NBR_Prototype также хранят значения кратности дуг и расположения токенов по порядковым номерам мест и переходов в соответствующих списках имен элементов шаблона.

Для быстрого выбора шаблона из коллекции было введено понятие целочисленной *маски*, которая играет роль хеша, построенного по текстовому описанию ТД. Разработана следующая схема побитового кодирования маски. Младшие 8 бит хранят общее число мест ТД. Биты с 9 и старше содержат флаг участия места в операции. Порядковый номер бита, не считая младшие 8 бит, соответствует индексу в массиве Index, который, в свою очередь, хранит индекс на массив прототипа, содержащий имена соответствующих элементов. Таким образом, при операции сравнения, когда определяется, подходит шаблон для слияния или нет, достаточно сравнить только маску входной и выходной ТД. Стоимость операции сравнения двух чисел — $O(1)$. При слиянии получение имени элемента, который нужно присоединить из шаблона к основной сети, также составляет $O(1)$. Нужно из маски взять нужный бит, по нему в индексе Index достать порядковый номер элемента, по которому уже получить добавляемый элемент.

На рис. 10 представлен пример описания внутренних целочисленных списков, которые сокращают время добавления шаблона в генерируемую сеть. Нетрудно видеть, что для хранения значения маски для n мест требуется $2^n + k$ бит, где k — это число бит для хранения самого значения n. В настоящее время размер маски ограничен 32-битным целым числом, а вспомогательный массив имеет 32 элемента. Соответственно, в текущей реализации шаблоны могут сливаться не более чем по 32 разным местам. Данные ограничения являются временными и могут быть легко расширены на любое число элементов шаблона применением специальных битовых массивов для хранения последовательностей битов произвольной длины.

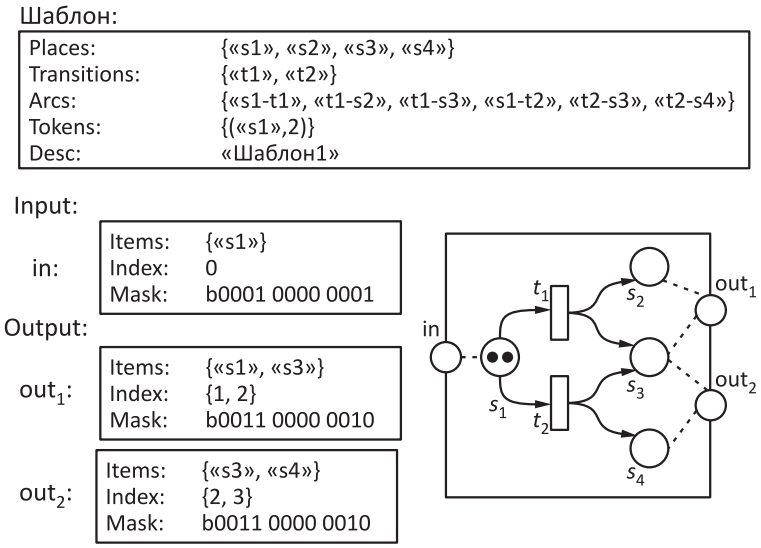

Рис. 10 Описание внутренних структур шаблона

Непосредственный выбор шаблона из коллекции по маске входной ТД происходит случайным образом. Для сокращения затрат на поиск подходящего шаблона по маске также применяется дополнительный индекс (сортировка по маске), который за $O(1)$ позволяет выбрать элемент из списка.

6 Анализ алгоритма

Проведем оценку сложности разработанного алгоритма. Алгоритм генерации СП по коллекции шаблонов является итерационным. На каждой итерации запускаются три потенциально ресурсоемкие процедуры: случайный выбор шаблона для шага эволюции коллекции, слияние стартового шаблона (результирующая сеть) с выбранным шаблоном и пополнение списка выходных точек стартового шаблона для последующих шагов эволюции.

Общеизвестно, что процедура извлечения головного элемента из списка выходных ТД AP реализуется за $O(1)$ [11], поэтому не будем ее рассматривать. Выше уже было отмечено, что за счет применения маски и индекса шаблонов по маске процедуру выбора можно реализовать за $O(1)$. Поэтому она также не влияет на общую скорость работы шага эволюции. Ее тоже не будем рассматривать. Слияние шаблонов на каждом шаге эволюции также реализуется за $O(n)$. Это обычный цикл по числу элементов, которые последовательно добавляются в стартовый шаблон. Дуги также переносятся за $O(n)$ за счет наличия индекса порядковых номеров элементов в шаблоне.

Остается процедура дополнения списка ТД стартового шаблона. Положим, имеется в наличии такая коллекция шаблонов, что в ней каждый шаблон имеет по n элементов в своей структуре и по m ТД в выходном интерфейсе. Тогда для общей оценки времени работы итерации получим $O(n + m^2)$. Возможны четыре варианта входных данных относительно N — числа элементов генерируемой сети:

(1) большие шаблоны с большим числом выходных ТД (число сравнимо с N);

(2) большие шаблоны с небольшим числом выходных ТД;

(3) небольшие шаблоны с большим числом выходных ТД;

(4) небольшие шаблоны с небольшим числом выходных ТД.

Варианты 1 и 3 приводят к $O(m^2)$. Все время уйдет на пополнение списка ТД, по которым еще возможна эволюция коллекции шаблонов. Вариант 2 приводит к $O(n)$. Число ТД в выходных интерфейсах несравнимо меньше, чем число элементов структуры шаблона, поэтому оценкой по m можно пренебречь. Вариант 4 более сложный. Когда число элементов в структуре шаблона сравнимо с числом элементов, которые надо сгенерировать, а соответственно и число ТД тоже сравнимо с N, оценка алгоритма будет приближаться к $O(n^2)$ (m и n — сравнимые величины). Если же $N \gg n$ и $n \sim m$, то оценка будет в худшем случае приближаться к $O(n^2)$. Если же $N \gg n$ и $n \gg m$, то приходим к варианту 2, когда оценкой по m можно пренебречь, и тогда получаем $O(n)$.

Таким образом, можно сказать, что в лучшем случае алгоритм имеет оценку $O(n)$, в худшем случае — $O(n^2)$. Отметим, что варианты с большим числом выходных ТД вряд ли имеют какое-то практическое значение с точки зрения получаемой на выходе сети, а также трудоемкости подготовки входных данных (описания ТД у шаблонов). Для «хороших» входных данных, когда число ТД небольшое по сравнению с числом элементов в структуре шаблонов и число элементов в результирующей сети несравнимо больше числа элементов в шаблоне, оценка алгоритма будет оптимальной — $O(n)$.

7 Тестирование программной реализации

Рассмотрим на простом примере результаты работы предложенного алгоритма для сетей различной величины. В качестве набора шаблонов возьмем шаблоны, описывающие алгоритмические конструкции императивного языка программирования. Будем использовать общепринятое представление потока управления этих конструкций в СП [12]. Таким образом, в результате генерации предполагается получить сеть, которая имитирует поведение последовательного алгоритма, состоящего из условий, циклов и последовательных конструкций.

Для тестирования времени работы алгоритма использовался один и тот же набор шаблонов для генерации сетей размером от 10^2 до 10^7 элементов. Результаты расчетов приведены в таблице.

Результаты измерений скорости работы алгоритма генерации

Число элементов, шт	Время работы, с	Объем памяти, МБ
100	0,001	14
500	0,003	15
1 000	0,024	15,5
2 000	0,045	16
3 000	0,068	16,5
4 000	0,093	17
5 000	0,117	18
6 000	0,142	18,5
7 000	0,162	19
8 000	0,183	19,5
9 000	0,215	20
10 000	0,234	20,5
50 000	1,277	43
100 000	2,567	72
500 000	13,521	311
1 000 000	29,108	607
10 000 000	319,494	6000

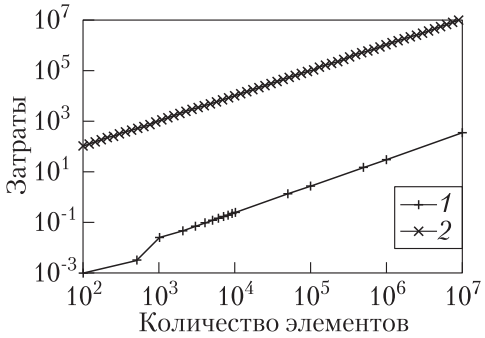

Рис. 11 График производительности алгоритма генерации: *1* — алгоритм генерации; *2* — функция $O(n)$

На рис. 11 представлен график произвольности: *1* — это результаты измерений работы алгоритма; *2* — это линейная функция $O(n)$. Отметим, что шкалы по осям OX и OY являются логарифмическими. Видно, что после 1000 элементов алгоритм ведет себя стабильно линейно. До 1000 элементов наблюдается квадратичная зависимость от числа элементов результирующей сети.

Вместе со скоростью работы алгоритма оценивались затраты на использование оперативной памяти. Они также растут линейно в зависимости от объема генерируемой сети. Это показывает, что затраты на применение индексов являются минимальными по сравнению с объемом памяти, занимаемым самой результирующей сетью.

8 Заключение

В общем случае проблема анализа программ включает две совершенно разные задачи. Первая задача состоит в построении пригодной для анализа модели программы. Вторая — в применении к этой модели методов анализа. Обе эти задачи являются сложными, но по разным причинам. Если в случае применения методов анализа основную проблему представляет комбинаторный взрыв числа состояний анализируемой модели, то в первом случае эта проблема заключается в отсутствии системного подхода к автоматизации построения моделей программ и, как следствие, отсутствии автоматических средств построения этих моделей. В результате требуется вмешательство человека для того, чтобы анализируемая модель содержала все необходимые события и состояния программы и не содержала излишних событий и состояний для сокращения требований к вычислительным ресурсам при ее анализе. Такая ситуация тормозит развитие как средств анализа, так и средств

построения моделей. Предложенный в настоящей статье подход предназначен для ускорения развития методов анализа за счет подготовки первоначального множества моделей, на которых эти средства будут тестироваться, причем в качестве ориентира выбраны большие модели, требующие оптимизации алгоритмов относительно производительности и потребляемой памяти, а также адаптации алгоритмов для работы на многопроцессорных системах. Кроме того, необходима разработка новых подходов по представлению результатов анализа. В работе представлен простой генератор СП, позволяющий имитировать процесс формирования императивной программы из стандартных алгоритмических конструкций. Тестирование генератора показало его работоспособность и практически линейную зависимость производительности и требований к объему памяти относительно размера генерируемой сети.

Литература

1. *Jensen K., Kristensen L. M., Mailund T.* The sweep-line state space exploration method // Theor. Comput. Sci., 2012. Vol. 429. P. 169–179. doi: 10.1016/j.tcs.2011.12.036.
2. *Abid C. A., Zouari B.* Local verification using a distributed state space // Fundam. Inform., 2013. Vol. 125. No. 1. P. 1–20. doi: 10.3233/FI-2013-850.
3. *Jezequel L., Fabre E., Khomenko V.* Factored planning: From automata to Petri nets // ACM T. Embed. Comput. S., 2015. Vol. 14. No. 2. Article No. 26. doi: 10.1145/2656215.
4. *Kammoun M. A., Rezg N., Achour Z., Rezig S.* State space search for safe time Petri nets based on binary decision diagrams tools: Application to air traffic flow management problem // Stud. Inform. Control, 2016. Vol. 25. No. 1. P. 39–50. doi: 10.24846/v25i1y201605.
5. *Wang C., Tao Y., Zhou Y.* Protocol verification by simultaneous reachability graph // IEEE Commun. Lett., 2017. Vol. 21. No. 8. P. 1727–1730. doi: 10.1109/LCOMM.2017.2695191.
6. *Thierry-Mieg Y.* Symbolic model-checking using ITS-tools // 21st Conference (International) on Tools and Algorithms for the Construction and Analysis of Systems Proceedings. — London: Springer, 2015. P. 231–237.
7. *Amparore E. G., Beccuti M., Donatelli S.* (Stochastic) model checking in GreatSPN // Application and theory of Petri nets and concurrency / Eds. G. Ciardo, E. Kindler. — Lecture notes in computer science ser. — Springer, 2014. Vol. 8489. P. 354–363. doi: 10.1007/978-3-319-07734-5_19.
8. *Berthomieu B., Dal Zilio S., Fronc L.* Model-checking real-time properties of an aircraft landing gear system using fiacre // Commun. Com. Inf. Sc., 2014. Vol. 433. P. 110–125. doi: 10.1007/978-3-319-07512-9_8.
9. *Wolf K.* Running LoLA 2.0 in a model checking competition // Transactions on Petri nets and other models of concurrency XI / Eds. M. Koutny, J. Desel, J. Kleijn. — Lecture notes in computer science ser. — Berlin–Heidelberg: Springer-Verlag, 2016. Vol. 9930. P. 274–285. doi: 10.1007/978-3-662-53401-4_13.
10. *Харитонов Д. И., Голенков Е. А., Тарасов Г. В., Леонтьев Д. В.* Метод генерации примеров моделей программ в терминах сетей Петри // Моделирование и анализ

информационных систем, 2015. Т. 22. № 4. С. 563–577. doi: 10.18255/1818-1015-2015-4-563-577.

11. *Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. И., Штайн К.* Алгоритмы: построение и анализ / Пер. с англ. — 2-е изд. — М.: Вильямс, 2005. 1296 с. (*Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to algorithms. — 2nd ed. — Cambridge, MA, USA: MIT Press; Boston: McGraw-Hill, 2001. 1184 p.)
12. *Питерсон Дж. Л.* Теория сетей Петри и моделирование систем / Пер. с англ. — М.: Мир, 1984. 264 с. (*Peterson J. L.* Petri net theory and the modeling of systems. — Englewood Cliffs, NJ, USA: Prentice-Hall, 1981. 241 p.)

Поступила в редакцию 10.10.17

SOFTWARE IMPLEMENTATION OF A METHOD FOR GENERATION OF PETRI NETS OF LARGE SIZE

D. I. Kharitonov, G. V. Tarasov, and D. V. Leontyev

Institute of Automation and Control Processes, Far-Eastern Branch of the Russian Academy of Sciences, 5 Radio Str., Vladivostok 690041, Russian Federation

Abstract: The software implementation of a method for generation of Petri nets having millions of elements of places and transitions is described. Particular attention is paid to the internal data structures and algorithmic complexity of the method. The presented material allowed the authors to obtain a software tool with computational complexity $O(n)$ where n is the number of elements in a model. The described theoretical results are justified by the performance tests in practical experiments.

Keywords: Petri nets; program model; data structures; algorithm analysis; object-oriented programming

DOI: 10.14357/08696527180208

Acknowledgments

The research was supported by the Program of the Presidium of the Russian Academy of Sciences (project 0262-2014-0157) and by the Far East Program (project 15-I-4-025).

References

1. Jensen, K., L. M. Kristensen, and T. Mailund. 2012. The sweep-line state space exploration method. *Theor. Comput. Sci*. 429:169–179. doi: 10.1016/ j.tcs.2011.12.036.
2. Abid, C. A., and B. Zouari. 2013. Local verification using a distributed state space. *Fundam. Inform.* 125(1):1–20. doi: 10.3233/FI-2013-850.

3. Jezequel, L., E. Fabre, and V. Khomenko. 2015. Factored planning: From automata to Petri nets. *ACM T. Embed. Comput. S.* 14(2). Article No. 26. doi: 10.1145/2656215.
4. Kammoun, M. A., N. Rezg, Z. Achour, and S. Rezig. 2016. State space search for safe time Petri nets based on binary decision diagrams tools: Application to air traffic flow management problem. *Stud. Inform. Control* 25(1):39–50. doi: 10.24846/v25i1y201605.
5. Wang, C., Y. Tao, and Y. Zhou. 2017. Protocol verification by simultaneous reachability graph. *IEEE Commun. Lett.* 21(8):1727–1730. doi: 10.1109/LCOMM.2017.2695191.
6. Thierry-Mieg, Y. 2015. Symbolic model-checking using ITS-tools. *21st Conference (International) on Tools and Algorithms for the Construction and Analysis of Systems Proceedings*. London: Springer. 231–237.
7. Amparore, E. G., M. Beccuti, and S. Donatelli. 2014. (Stochastic) model checking in GreatSPN. *Application and theory of Petri nets and concurrency*. Eds. G. Ciardo and E. Kindler. Lecture notes in computer science ser. 8489:354–363. doi: 10.1007/978-3-319-07734-5_19.
8. Berthomieu, B., S. Dal Zilio, and L. Fronc. 2014. Model-checking real-time properties of an aircraft landing gear system using fiacre. *Comm. Com. Inf. Sc.* 433:110–125. doi: 10.1007/978-3-319-07512-9_8.
9. Wolf, K. 2016. Running LoLA 2.0 in a model checking competition. *Transactions on Petri nets and other models of concurrency XI*. Eds. M. Koutny, J. Desel, and J. Kleijn. Lecture notes in computer science ser. Berlin–Heidelberg: Springer-Verlag. 9930:274–285. doi: 10.1007/978-3-662-53401-4_13.
10. Kharitonov, D. I., E. A. Golenkov, G. V. Tarasov, and D. V. Leontyev. 2015. Metod generatsii primerov modeley programm v terminakh setey Petri [A method of sample models of program construction in terms of Petri nets]. *Modelirovanie i analiz informatsionnykh sistem* [Modeling and Analysis of Information Systems] 22(4):563–577. doi: 10.18255/1818-1015-2015-4-563-577.
11. Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. 2001. *Introduction to algorithms*. 2nd ed. Cambridge, MA: MIT Press; Boston: McGraw-Hill. 1184 p.
12. Peterson, J. L. 1981. *Petri net theory and the modeling of systems*. Englewood Cliffs, NJ: Prentice-Hall. 241 p.

Received October 10, 2017

Contributors

Kharitonov Dmitriy I. (b. 1973) — Candidate of Science (PhD) in technology, senior scientist, Institute of Automation and Control Processes, Far-Eastern Branch of the Russian Academy of Sciences, 5 Radio Str., Vladivostok 690041, Russian Federation; demiurg@dvo.ru

Tarasov Georgiy V. (b. 1977) — scientist, Institute of Automation and Control Processes, Far-Eastern Branch of the Russian Academy of Sciences, 5 Radio Str., Vladivostok 690041, Russian Federation; george@dvo.ru

Leontyev Denis V. (b. 1992) — PhD student, Institute of Automation and Control Processes, Far-Eastern Branch of the Russian Academy of Sciences, 5 Radio Str., Vladivostok 690041, Russian Federation; devozh@dvo.ru