МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий

Кафедра систем информатики

Направление подготовки: 230100 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

Магистерская программа: технология разработки программных систем

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

# Применение раскрашенных сетей Петри для верификации спецификаций коммуникационных протоколов

Стененко Александр Александрович

Тема диссертации утверждена распоряжением по НГУ № 532 от «14» декабря 2012г.

Тема диссертации скорректирована распоряжением по НГУ № 111 от «20» марта 2014г.

**«К защите допущена»**

Заведующий кафедрой,

д. ф.-м. н., профессор

Лаврентьев М. М. / _____

(фамилия , И., О.) / (подпись, МП)

«____» _____ 2014г.

**Научный руководитель**

Зав. лаб., ИСИ СО РАН

к. ф.-м. н., с. н. с.

Непомнящий В. А. / _____

(фамилия , И., О.) / (подпись, МП)

«____» _____ 2014г.

Дата защиты: «____» _____ 2014г.

Автор Стененко А. А. / _____

(фамилия , И., О.) / (подпись)

Новосибирск, 2014 г.

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий

Кафедра систем информатики

Направление подготовки: 230100 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

Магистерская программа: технология разработки программных систем

УТВЕРЖДАЮ

Зав. Кафедрой Лаврентьев М. М.
(фамилия, И., О.)

_____
(подпись, МП)
«___» _____ 2014 г.

**ЗАДАНИЕ**

**НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ**

**МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ**

Студенту Стененко Александру Александровичу

Тема: применение раскрашенных сетей Петри для верификации спецификаций коммуникационных протоколов

Исходные данные (или цель работы):

Требуется создать эффективный транслятор из языка раскрашенных сетей Петри (РСП) в язык Promela, а также транслятор из языка автоматных спецификаций в РСП. С использованием созданных инструментов требуется провести эксперименты по верификации моделей коммуникационных систем.

Структурные части работы

- Анализ языков описания моделей (язык автоматных спецификаций, РСП и Promela).
- Введение ограничений на входную модель.
- Разработка алгоритма трансляции из РСП в язык Promela.
- Реализация транслятора из РСП в язык Promela.
- Реализация транслятора из языка автоматных спецификаций в РСП.
- Верификация моделей коммуникационных систем с использованием созданных инструментов.

Научный руководитель
Зав. лаб, ИСИ СО РАН,
к. ф.-м. н., с. н. с.
Непомнящий В. А. / _____
(фамилия , И., О.) / (подпись)
«__» _____ ____ г.

Задание принял к исполнению
Стененко А. А. / _____
(ФИО студента) / (подпись)
«__» _____ ____ г.

# Оглавление

# ВВЕДЕНИЕ

Распределённые вычислительные системы, применяемые во многих областях человеческой деятельности, используют различные телекоммуникационные протоколы. Актуальной задачей является изучение свойств и проверка корректности этих протоколов (их верификация).

Протоколы могут быть формально описаны на разных языках, одним из которых является язык автоматных спецификаций [1]. Модели, представленные на данном языке, лаконичны и достаточно просты в описании. Язык раскрашенных сетей Петри (РСП) [2] обладает большой выразительностью и является широко используемым для моделирования взаимодействия происходящего в распределённых системах формализмом [3-5]. Для анализа моделей, представленных в виде РСП, созданы различные средства, например система CPN Tools [6]. Для верификации таких моделей методом проверки моделей (model checking) существует система Helena [7], которая позволяет проверять свойства, представленные формулами линейной темпоральной логики (LTL), но не поддерживает РСП, в которых используются временные конструкции. Для средств верификации РСП, таких как Neco [8] и других [9] не описаны ограничения на входные модели, и в частности не рассматривается применение этих систем для верификации РСП, использующих временные конструкции. Таким образом, верификация свойств РСП с временными конструкциями является открытой проблемой.

Язык Promela, являющийся входным для системы верификации SPIN [10, 11], имеет черты, присущие языкам программирования. Для верификации SPIN использует метод проверки моделей, состоящий в полном обходе графа состояний модели [12]. Эта система позволяет проверять LTL-свойства.

Цель работы состоит в создании системы верификации моделей, представленных как на языке автоматных спецификаций, так и на языке РСП. Для этого требуется решить следующие задачи:

- разработать транслятор из автоматных спецификаций в РСП, реализующий алгоритм трансляции, описанный в [1];
- разработать алгоритм трансляции РСП в язык Promela, при этом указать, каким ограничениям должны удовлетворять РСП-модели, чтобы их было возможно транслировать в язык Promela разработанным алгоритмом;
- разработать транслятор из РСП в язык Promela, реализующий разработанный алгоритм;
- провести эксперименты по верификации моделей распределённых систем с помощью

разработанных средств для апробации выбранного подхода.

Для эффективности верификации требуется, чтобы пространство достижимых состояний модели, полученной в результате трансляции, существенно не увеличивалось по сравнению с пространством достижимых состояний исходной модели.

# ГЛАВА 1. Предварительные понятия

## 1.1 Раскрашенные сети Петри

Различные виды сетей Петри широко используются для моделирования систем, в которых присутствуют несколько взаимодействующих процессов. Сетевая модель представляет собой двудольный ориентированный граф, вершины одной доли которого называются *местами*, а другой — *переходами*. Состояние модели определяется *разметкой мест*: в каждом из мест сети находится некоторое количество *фишек*. Переходы и дуги определяют поведение модели: переход *может сработать* если в его входных местах находится достаточное количество фишек. При *срабатывании* перехода из его входных мест фишки извлекаются, а в его выходные места — помещаются. Раскрашенными сети Петри называются в том случае, если фишки являются значениями некоторых типов данных, которые принято называть *наборами цветов*. В такой сети дугам приписаны выражения. При срабатывании переходов значения выражений на дугах вычисляются. Результаты вычислений извлекаются из разметки входных мест перехода и помещаются в разметку выходных мест. Переходам могут быть приписаны *охранные* выражения. Если охранное выражение принимает значение «ложь» то срабатывание перехода запрещается. В этой главе приводится формальное определение раскрашенных сетей Петри, взятое из [2].

Пусть S = {$s_1$ , $s_2$ , $s_3$ , … } — непустое множество. **Мультимножество над S** — это функция m : S → N, отображающая каждый элемент s ∈ S в неотрицательное целое число. m(s) ∈ N называется **количеством вхождений** (коэффициентом) элемента s в m.

Мультимножество m также может быть представлено в виде суммы:

$$^{++}\sum_{s\in S} m(s)\grave{}\,s = m(s_1)\grave{}\,s_1 ++ m(s_2)\grave{}\,s_2 ++ m(s_3)\grave{}\,s_3 ++ \ldots$$

Где запись $m = n\grave{}\,s_i$ означает, что $m(s_k) = \{\, n \text{, если } i = k \text{; } 0 \text{ в противном случае} \,\}$

**Принадлежность**, **сложение**, **скалярное умножение**, **сравнение** и **размер** определены для мультимножеств следующим образом:

пусть $m_1$, $m_2$, и m — мультимножеста и n ∈ N, тогда

1. $\forall s \in S : s \in m \Leftrightarrow m(s) > 0.$
2. $\forall s \in S : (m_1 ++ m_2)(s) \Leftrightarrow m_1(s) + m_2(s).$
3. $\forall s \in S : (n ** m)(s) \Leftrightarrow n * m(s).$
4. $m_1 <<= m_2 \Leftrightarrow \forall s \in S : m_1(s) \le m_2(s).$
5. $|m| = \sum_{s\in S} m(s).$

Если $m_1 <<= m_2$, то **вычитание** определено следующим образом:

6. $\forall s \in S : (m_1 -- m_2)(s) = m_1(s) - m_2(s)$.

Множество всех мультимножеств над S обозначается $S_{MS}$. Пустое мультимножество над S обозначается $ø_{MS}$ и определяется как $ø_{MS}(s) = 0$ для всех $s \in S$.

Пусть $EXPR_v$ — множество выражений, в которых используются переменные из множества v. **Раскрашенная сеть Петри** — это кортеж
(P, T, A, Σ, V, C, G, E, I), где:

1. P — конечное множество **мест**.
2. T — конечное множество **переходов**, такое что $P \cap T = ø$.
3. $A \subseteq P \times T \cup T \times P$ — множество направленных **дуг**.
4. Σ — конечное множество непустых **«наборов цветов»** (типов данных).
5. V — конечное множество **типизированных переменных**, таких что $Type[v] \in \Sigma$ для всех переменных $v \in V$ .
6. $C : P \rightarrow \Sigma$ — **функция наборов цветов**, присваивающая набор цветов каждому месту.
7. $G : T \rightarrow EXPR_V$ — **функция охранных выражений**, присваивающая охранное выражение каждому переходу; $Type[G(t)] = Bool$.
8. $E : A \rightarrow EXPR_V$ — **функция выражений на дугах**, присваивающая выражение каждой дуге; $Type[E(a)] = C(p)_{MS}$ , где p — место инцидентное дуге a.
1. $I : P \rightarrow EXPR_ø$ — **функция инициализации**, присваивающая инициализирующее выражение каждому месту p; $Type[I(p)] = C(p)_{MS}$.

Для раскрашенной сети Петри CPN = (P, T, A, Σ, V, C, G, E, I) определим следующие понятия:

1. **Разметка** — это функция M, отображающая каждое место $p \in P$ в мультимножество меток $M(p) \in C(p)_{MS}$ .

2. **Начальная разметка** $M_0$ определяется как $M_0(p) = I(p)$ для всех $p \in P$.

3. **Переменные перехода** t обозначаются как $Var(t) \subseteq V$ и состоят из свободных переменных встречающихся в охранной функции перехода t и в выражениях, приписанных дугам инцидентным переходу t.

4. **Связывание переменных перехода** t — это функция b, отображающая каждую переменную $v \in Var(t)$ в значение $b(v) \in Type[v]$. Множество связываний перехода t обозначается B(t).

5. Э**лемент связывания** — это пара (t, b), такая что $t \in T$ и $b \in B(t)$. Множество всех элементов связывания BE(t) перехода t определяется как $BE(t) = \{ (t, b) \mid b \in B(t) \}$.

Множество всех элементов связывания модели обозначается BE.

6. **Шаг** $Y \in BE_{MS}$ — это непустое конечное мультимножество связываний элементов.

Элемент связывания (t, b) ∈ BE является **допустимым** при разметке M, если и только если следующие два свойства выполнены:

1. G(t)<b>.
2. ∀p ∈ P : E(p, t)<b> <<= M(p).

Если элемент связывания (t, b) является допустимым при разметке M, то он может быть **осуществлен**, при этом сеть переходит в состояние с разметкой M', определённой следующим образом:

3. ∀p ∈ P : M'(p) = (M(p) −− E(p, t)<b>) ++ E(t, p)<b>.

Шаг $Y \in BE_{MS}$ является **допустимым** при разметке M, если и только если следующие два свойства выполнены:

1. $\forall (t,b) \in Y : G(t) < b >$
2. $\forall\, p \in P : \overset{++}{\underset{MS}{\sum}}\limits_{(t,b)\in Y} E(p,t) < b > <<= M(p)$

Здесь и далее символ $\overset{++}{\underset{MS}{\sum}}$ обозначает сумму мультимножеств.

Если шаг Y является допустимым при разметке M, то он может быть **осуществлён**, при это сеть переходит в состояние с разметкой M', определённой следующим образом:

3. $\forall\, p \in P : M'(p) = (M(p) -- \overset{++}{\underset{MS}{\sum}}\limits_{(t,b)\in Y} E(p,t) < b >) ++ \overset{++}{\underset{MS}{\sum}}\limits_{(t,b)\in Y} E(p,t) < b >$

**Конечная последовательность исполнения длины** n (n≥0) — это последовательность чередующихся разметок и шагов, записывающаяся как $M_1 \xrightarrow{Y1} M_2 \xrightarrow{Y2} M_3 \cdots M_n \xrightarrow{Yn} M_{n+1}$, такая что $M_i \xrightarrow{Yi} M_{i+1}$ для всех $1 \le i \le n$. Все разметки в последовательности являются **достижимыми** из $M_1$. Из этого следует, что любая разметка M является достижимой из самой себя в результате тривиальной последовательности исполнения длины 0.

Аналогично, **бесконечная последовательность исполнения** — это последовательность разметок и шагов $M_1 \xrightarrow{Y1} M_2 \xrightarrow{Y2} M_3 \xrightarrow{Y3} \cdots$, такая что $M_i \xrightarrow{Yi} M_{i+1}$ для всех $i \ge 1$. Множество разметок достижимых из разметки M обозначается как R(M). Множество **достижимых разметок** — это $R(M_0)$, то есть множество разметок, достижимых из начальной разметки $M_0$.

Понятие времени в РСП определяется с помощью *временных фишек*. Временная фишка принимает значения *временного типа данных*: помимо цвета она имеет также временную *метку* — значение момента времени, начиная с которого она становится

доступна для использования переходами. Для моделей с временными конструкциями состояние определяется не только разметкой мест, но и текущим моментом времени. Значение текущего момента вместе с временной меткой фишки определяет доступна ли данная фишка в текущем состоянии. Увеличение момента времени при исполнении модели происходит только в том случае, если ни один из переходов не может сработать, если момент не будет увеличен.

С целью проведения верификации сетевых моделей с помощью системы SPIN, на них накладываются следующие ограничения:

- мультимножества и списки финитны, то есть для каждого из соответствующих типов данных установлено максимальное количество элементов, которые могут в них содержаться;
- поддерживаемые типы данных: INT, BOOL, UNIT (вырожденный тип — пустой кортеж), перечисления, кортежи и списки.

Кроме того требуется, чтобы выражения на входных дугах переходов позволяли производить перебор значений переменных переходов, исходя из разметки входных мест переходов.

Из ограничения финитности следует изменение определения **допустимости**: элемент связывания (t, b) ∈ BE является допустимым при разметке M, если и только если следующие свойства выполнены:

1. G(t)<b>.
2. ∀p ∈ P : E(p, t)<b> <<= M(p).
3. ∀p ∈ P : *valid_value*( (M(p) -- E(p, t)<b>) ++ E(t, p)<b> ), где *valid_value*(v) — истина, если и только если мультимножество v помещается в отведённую память.

В отличие от моделей CPN Tools, где для каждого из состояний определён текущий момент времени, значение которого показывает количество временных тактов, прошедших с начала исполнения модели, здесь на понятие времени наложено ограничение «относительности». Это означает, что величины временных меток у фишек рассматриваются не относительно абсолютного «нулевого» момента, например, начала исполнения модели, а относительно текущего момента времени, а сама величина текущего момента не входит в состояние модели. Это позволяет считать состояния, отличающиеся лишь сдвигом во времени меток всех фишек, одинаковыми. Увеличение значения глобального для модели счётчика тактов времени может быть заменено уменьшением значений временных меток всех фишек, присутствующих в модели. Метки всегда

неотрицательны. Значение метки n показывает, что фишка будет доступна через n тактов времени после текущего момента. Нулевые метки времени соответствуют текущему моменту. Данное ограничение запрещает использование оператора, указывающего абсолютный момент времени «@» в выражениях на дугах, однако этот оператор может использоваться в выражениях, задающих начальную разметку сети. В качестве аргумента оператора времени, показывающего временную задержку, должны быть использованы неотрицательные числа. Так как доступность фишки в некотором состоянии определяется разностью значения её временной метки и значения текущего момента времени, то вместо того, чтобы увеличивать значение текущего момента, можно уменьшить значения временных меток всех присутствующих в модели фишек, а значение текущего момента считать нулевым во всех состояниях. Следовательно, поведение сетевой модели, удовлетворяющей указанному ограничению совпадает с поведением модели в CPN Tools.

РСП-модель может быть иерархической. В таком случае она включает в себя несколько уровней иерархии РСП, один из которых является корневым, а все остальные могут быть вложены в другие, с помощью так называемых *переходов-подстановок*. Вложенность не может быть циклической. При функционировании иерархической РСП переход-подстановка не срабатывает как обычный переход, а ведёт себя так, как если бы на его месте находился подуровень иерархии РСП, который он представляет. Иерархические РСП не увеличивают выразительность языка РСП, но позволяют облегчить моделирование сложных систем с использованием абстрагирования. Кроме того места РСП, имеющие одинаковый тип и одинаковую начальную разметку, могут быть объединены логически в одно место с помощью определения *множеств слияний* мест. Все места, входящие в одно множество-слияние имеют одинаковую разметку в любом состоянии, при изменении разметки одного из таких мест, аналогичным образом меняется разметка остальных мест. Использование множеств-слияний также не увеличивает выразительность языка, но может быть полезно для объединения мест принадлежащих разным уровням иерархии в одно место РСП.

### 1.2 Системы взаимодействующих расширенных конечных автоматов

Некоторые распределённые системы удобно моделировать в терминах взаимодействия конечных автоматов. В том случае, если при срабатывании переходов автомат может изменять значения переменных, присутствующих в системе, а допустимость переходов зависит не только от состояния автомата, но и от значений переменных, можно говорить про расширенные конечные автоматы. Взаимодействие

автоматов происходит с помощью асинхронного обмена сообщениями. В этой главе приводится формальное определение систем взаимодействующих расширенных конечных автоматов взятое из [1].

**Расширенный конечный автомат (РКА) есть кортеж α =<S, V, G, I, O, T>, где**

1. S – набор **состояний** автомата;
2. V – набор **локальных переменных** автомата, включающий специальную переменную id –идентификатор автомата, используемый для взаимодействия с другими автоматами;
3. G – набор **глобальных переменных**;
4. I – набор **входных сигналов**;
5. O – набор **выходных сигналов**.

   Входные и выходные сигналы имеют формат <Src, Dest, Type, Param>, где Src – это идентификатор отправителя, Dest – идентификатор получателя, Type – перечислимый тип сигнала и Param – параметр сигнала.
6. T – набор **переходов**. Каждый переход из – это кортеж

   t = {Ss, Se,Os(G, V, I), P(G, V, I), E(G, V, I)}, где

1. Ss – **исходное состояние**;
2. Se – **результирующее состояние**;
3. I – **входной сигнал** из окружения Env (может отсутствовать, тогда срабатывание перехода не зависит от входных сигналов);
4. Os(G, V, I) – множество **выходных сигналов**, которое является подмножеством O;
5. P (G, V, I) – предикат, который задает **условие срабатывания перехода**;
6. E(G, V, I) – множество операторов присваивания, которые задают **вычисления автомата**, используя локальные и глобальные переменные.

**Система взаимодействующих РКА** – это кортеж Σ =<A, G, Env, Init>, где

1. A – множество **расширенных конечных автоматов**;
2. G – множество **глобальных переменных** всех автоматов в системе;
3. Env – окружение, которое является множеством сигналов в системе, ожидающих приёма;
4. Init – **инициализирующая функция**, определённая на G и A, которая задаёт начальные состояния для всех РКА из множества A, начальные значения локальных и глобальных переменных, а также начальное множество сигналов Env.

Все множества в этих определениях должны быть конечными, а множество глобальных переменных не пересекается с множествами локальных переменных

автоматов.

**Конфигурацией автомата** α системы Σ на шаге k называется пара <S(k), V(k)>, где S(k) – **состояние** автомата, а V(k) – **значения** его **локальных переменных** на шаге k. **Конфигурация системы** на шаге k – это множество конфигураций всех автоматов системы, множество значений глобальных переменных и значение переменной Env на данном шаге.

Пусть автомат α системы Σ находится в состоянии Ss. **Переход** t автомата α **разрешён**, если значение P(G, V, I) истинно для некоторого сигнала I из Env.

**Срабатывание разрешённого перехода** означает:

- изменение состояния Ss на Se;
- удаление входного сигнала I из окружения Env, если I задан;
- изменение значений переменных согласно выражению E(G, V, I);
- помещение выходных сигналов Os(G,V, I) в окружение Env.

**Автомат срабатывает**, если срабатывает по крайней мере один из его разрешённых переходов (недетерминированный выбор). **Система срабатывает** на шаге k, если на этом шаге какой-либо автомат в системе срабатывает.

В автоматных спецификациях многих систем нередко присутствует несколько экземпляров автоматов, которые можно отнести к одному классу, возможно отличающихся друг от друга лишь значениями локальных переменных и состоянием. Чтобы не повторять описание нескольких почти одинаковых автоматов в спецификации классы автоматов будем задавать отдельно, а для задания каждого из экземпляров автоматов будем указывать его класс, значение его идентификатора и начальное состояние, а также начальные значения локальных переменных, если они отличаются от стандартных начальных значений для данного класса автоматов.

Если ёмкость среды конечна, то модель, представленная в виде системы взаимодействующих РКА, может быть транслирована в модель на языке РСП. Доказана корректность алгоритма трансляции [1].

### 1.3 Система верификации SPIN, использующая метод проверки моделей

При верификации методом проверки моделей (model checking) [12] происходит полный обход графа достижимых состояний модели. В процессе такого обхода верификатор сохраняет в памяти посещённые состояния во избежание зацикливания обхода. При недетерминированном выборе верификатор выполняет обход каждой из возможных ветвей исполнения. Верификатор завершает работу после того как будет

произведён полный обход графа состояний модели, либо если в процессе обхода будет обнаружена ошибка. Результатом его работы является либо сообщение об отсутствии ошибок в модели, либо контрпример — последовательность выполнения модели, приводящая к ошибке. Данный метод позволяет проверять выполнение для модели различных свойств: отсутствие тупиков, утверждения (assertions), а также свойства, выраженные формулами темпоральной логики.

Система верификации SPIN [4, 5] реализует метод проверки моделей и позволяет проверять свойства, выраженные на языке линейной темпоральной логики (LTL). Входной язык Promela системы верификации SPIN обладает синтаксисом, похожим на синтаксис современных языков программирования высокого уровня, таких как язык C. Основным понятием языка Promela является процесс, представляющий поток управления в модели. Код процесса состоит из конструкций таких, как присваивание значений переменным, операторы условного и недетерминированного выбора, атомарно исполняющиеся блоки (atomic), операторы перехода на метки (goto), утверждения (assert). Также в модели на языке Promela могут использоваться вставки кода на языке C. Функции и процедуры в самом языке Promela отсутствуют, но внутри вставок кода на языке C могут быть вызваны функции языка C. В модели могут использоваться переменные целочисленных типов, типа-перечисления, массивы, а также переменные языка C, в том числе структуры. Динамическая память не может быть использована. Вектор состояния модели образуют переменные и указатели на текущую инструкцию для каждого из процессов модели, то есть значения переменных и указателей на инструкции определяют состояние модели. Промежуточные состояния могут быть исключены из графа состояний модели с помощью заключения последовательности инструкций в атомарно исполняющийся блок. Блок кода на языке C также не будет требовать рассмотрения верификатором промежуточных состояний, возникающих при его выполнении.

# ГЛАВА 2. Верификация раскрашенных сетей Петри с помощью системы SPIN

Чтобы верифицировать РСП-модели с помощью системы SPIN, используется транслятор раскрашенных сетей Петри в язык Promela. Использование данного транслятора вместе с транслятором, описанным в третьей главе, позволяет также проводить верификацию автоматных спецификаций.

Для построения контрпримеров для сетевой модели транслятор добавляет в модель код, который записывает, какие переходы сработали, и какие значения при этом имели переменные.

## 2.1 Схема трансляции раскрашенных сетей Петри в язык Promela

Трансляция происходит поэтапно: сначала транслируются типы данных, затем переменные, места и переходы сети.

Состояния мест сети Петри представляются в выходной модели значениями входящих в вектор состояния Promela-модели переменных, а все переходы сети транслируются в код единственного процесса Promela-модели. Этот процесс содержит фрагмент кода на языке C, инициализирующий значения мультимножеств (он выполняется в начале исполнения модели) и фрагмент кода, в котором осуществляется поиск возможных переходов и их срабатывание. Код, осуществляющий поиск и срабатывание переходов, заключён в блок `atomic`. Согласно документации SPIN [10] последовательность операторов Promela, заключённая в такой блок, может содержать операторы недетерминированного выбора, при этом все промежуточные состояния модели, которые она проходит во время исполнения тела atomic-блока, исключаются из рассмотрения при верификации, в отличие от состояния, в которое переходит модель после выполнения всего блока.

На рис. 1 приводится общая схема модели на языке Promela, получающейся в результате трансляции РСП.

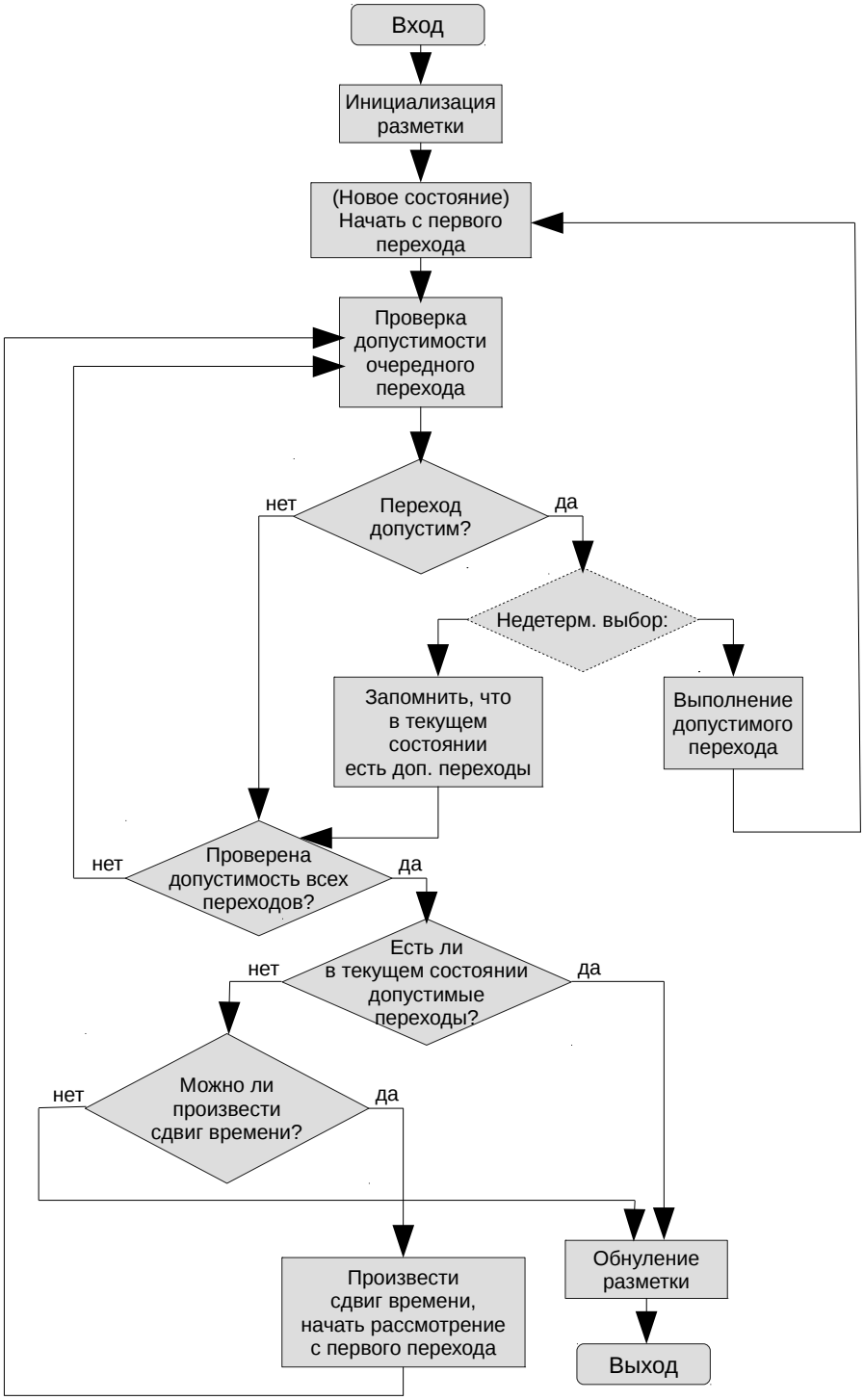


*Рис. 1. Общая схема функционирования выходной Promela-модели*

### 2.2 Трансляция типов данных

Типы данных сетей Петри транслируются в типы данных языка C. Целые числа, булевские, тип UNIT (пустой кортеж) и типы-перечисления транслируются в тип int языка C. Типы-кортежи транслируются в типы-структуры языка C, содержащие поля соответствующих типов. Типы-списки транслируются в типы-структуры языка C, содержащие два поля: массив для хранения элементов списка и длина списка. Для работы с каждым из типов данных определяются следующие функции языка C:

`cmp` — сравнивает значения двух переменных, возвращает отрицательный

результат если первая переменная меньше второй, положительный результат если первая переменная больше второй и ноль в случае равенства.

`null` — зануляет значение переменной.

Эти функции используются во вставках на языке C при выполнении модели на языке Promela.

Так как все типы-структуры языка C, представляющие списки, отличаются только названием и типом элементов списка, то для генерации определяющего их кода, а также кода функций для работы с ними используется шаблон, записанный с помощью макроса препроцессора языка C.

**Примеры определений типов данных.**

Определение типа данных `INT` (соответствует определению «`colset INT = int`»)

```
typedef int cs_int;
// функции для работы с данным типом:
int cmp_int(cs_int *a, cs_int *b) {
     return *a - *b;
}
void null_int(cs_int *a) {
     *a = 0;
}
```

Определение кортежа из двух целых чисел `IxI`, определённого как «`colset IxI = product int * int`»

```
typedef struct cs_IxI {
     cs_int field1;
     cs_int field2;
} cs_IxI;

// функции для работы с данным типом:
int cmp_IxI(cs_IxI *this, cs_IxI *that) {
     int res;
     res = cmp_int(&this->field1, &that->field1);
     if (res != 0) return res;
     res = cmp_int(&this->field2, &that->field2);
     return res;
}
void null_IxI(cs_IxI *val) {
     null_int(&val->field1);
     null_int(&val->field2);
}
```

### 2.3 Трансляция мест

Для представления мультимножеств, соответствующих разметке мест сети, используются переменные, имеющие тип-мультимножество. Этот тип является структурой

языка С, содержащей два поля: массив для хранения значений элементов и количество элементов в массиве.

Для работы с типами-мультимножествами определяются следующие функции:

`empty` — зануляет значение мультимножества.

`copy` — копирует элемент мультимножества, обращаясь к нему по индексу в массиве элементов.

`get` — извлекает элемент мультимножества, обращаясь к нему по индексу в массиве элементов.

`put` — добавляет элемент в мультимножество.

Данные функции обеспечивают сохранение следующих свойств:

- массив, содержащий элементы мультимножества, упорядочен;
- неиспользуемая часть массива заполняется нулевыми значениями.

Из этих свойств следует, что совпадение значений мультимножеств эквивалентно совпадению значений байтов памяти, отведённых для представления этих мультимножеств. Это позволяет верификатору по значению вектора состояния модели определять, было ли рассмотрено данное состояние модели, в результате чего количество достижимых состояний не увеличивается относительно исходной сетевой модели.

**Пример определения типа-мультимножества**

Определение типа данных, соответствующего мультимножеству над типом данных `IxI` (`IxI = product INT * INT`)

```
    typedef struct msOfIxI {
        int size;
        cs_IxI content[MS_MAX_SIZE];
    } msOfIxI;
// далее следуют функции для работы с мультимножеством:

//  присвоить мультимножеству ms
//  значение «пустое мультимножество»
    void empty_msOfIxI(msOfIxI *ms)

//  копировать значение i-ого элемента мультимножества ms в res
    void copy_msOfIxI(msOfIxI *ms, cs_IxI *res, int i)


//  извлечь из мультимножества ms i-ый элемент
//  и поместить его значение в res
    void get_msOfIxI(msOfIxI *ms, cs_IxI *res, int i)

//  поместить значение elem в мультимножество ms
```

```
    void put_msOfIxI(msOfIxI *ms, cs_IxI *elem)
```

Значения мультимножеств над типами данных, имеющими небольшое число возможных значений (такими типами считаются UNIT, BOOL и перечисления), представляются иначе. Для их представления используется набор целочисленных счётчиков количества вхождений в мультимножество каждого из значений типа данных.

**Пример определения типа-мультимножества над типом данных, имеющим небольшое число возможных значений**

Определение типа данных, соответствующего мультимножеству над типом данных `bool`:

```
    typedef struct ms1Ofbool {
        int count[2]; // массив счётчиков вхождений элементов
    } ms1Ofbool;
// далее определены функции для работы с мультимножеством
// над небольшим типом данных:

//  присвоить мультимножеству ms
//  значение «пустое мультимножество»
    void empty_ms1Ofbool(ms1Ofbool *ms)

//  извлечь из мультимножества ms элемент со значением elem
    void get_ms1Ofbool(ms1Ofbool *ms, cs_bool elem)

//  поместить элемент со значением elem в мультимножество ms
    void put_ms1Ofbool(ms1Ofbool *ms, cs_bool elem)

    // функция «copy» не определяется для ms1Ofbool,
    // так как проверка принадлежности элемента
    // этому мультимножеству производится
    // по значению счётчика вхождений count
```

Некоторые места в сети обладают тем свойством, что в любом достижимом состоянии их разметка содержит ровно один элемент. Разметка каждого из таких мест может быть представлена в модели Promela единственной переменной соответствующего типа, значение которой равно этому элементу единственному элементу мультимножества. Для того, чтобы проверить, обладает ли некоторое место сети Петри таким свойством, транслятор производит перебор всех переходов сети Петри, смежных с этим местом, и проверяет типы выражений на входящей и исходящей дугах, соединяющих место и переход.

Итак, перед трансляцией мест сети Петри определяются типы, представляющие мультимножества и функции для работы с ними. Так как эти типы отличаются только

названием и типом элементов мультимножества, то для генерации определяющего их кода, а также кода функций, работающих с ними, используются шаблоны, записанные с помощью макросов препроцессора языка C. После того как определены типы-мультимножества, для каждого места объявляется переменная, имеющая соответствующий тип.

### 2.4 Трансляция переходов

Сначала приведём описание того, как исполняется выходная модель на языке Promela. В первую очередь происходит инициализация переменных, соответствующих местам сети. После инициализации исполнение модели заключается в осуществлении цикла, называемого главным. Главный цикл состоит из двух этапов: поиск возможных переходов и срабатывание переходов.

Поиск возможных переходов реализуется следующим образом: для каждого перехода T просматриваются в цикле элементы мультимножеств, соответствующих входным местам перехода T и осуществляется поиск значений переменных перехода T, таких что выражения на дугах приняли бы соответствующие значения и охранное выражение стало бы истинным. Если такие значения переменных найдены, то происходит вычисление значений выражений на выходных дугах и производится проверка того, что ёмкость выходных мест перехода T достаточна, чтобы переход мог сработать. Если ёмкость достаточна, то переход считается возможным при вычисленных значениях переменных, эти значения сохраняются и недетерминированным образом производится выбор одной из альтернатив: либо переход осуществляется, либо продолжается поиск возможных переходов. Если поиск возможных переходов завершён и обнаружено, что ни один из переходов в текущем состоянии не возможен, то выполнение модели завершается. Осуществление переходов при вычисленных значениях переменных соответствует семантике раскрашенной сети Петри: из входных мест забираются вычисленные значения выражений на дугах, в выходные места — вычисленные значения выражений добавляются.

В качестве служебных переменных модели определяется необходимое количество целочисленных счётчиков `i1, i2, …` , используемых для перебора возможных значений выражений, приписанных входящим в переход дугам. Для запоминания таких значений этих счётчиков, при которых возможен переход, используются переменные `t_i1, t_i2, …` . Для запоминания номера возможного перехода используется служебная целочисленная переменная `trans`, переходы нумеруются с единицы, если ни один из переходов не

является возможным, то она принимает значение равное нулю.

Значения выражений, приписанных входным дугам, вычисляемые во время перебора входных значений мультимножеств присваиваются служебным переменным `fT<номер_перехода>_in<номер_дуги>`. Результаты вычислений выражений на дугах присваиваются служебным переменным:

`fT<номер_перехода>_in_v<номер_дуги>`,

`fT<номер_перехода>_out<номер_дуги>`.

Значение охранного выражения присваивается служебной переменной `guard`. Также используются следующие служебные переменные:

`bindingFail` — флаг, свидетельствующий, что невозможно произвести связывание значений переменных перехода.

`evaluationFail` — флаг, свидетельствующий, что невозможно произвести вычисление выражения.

`exprError` — флаг, свидетельствующий, что вычисление выражения приводит к ошибке (например, происходит деление на ноль или взятие головы пустого списка).

Переменные переходов раскрашенных сетей Петри транслируются в переменные языка С соответствующего типа.

**Пример определения переменных сетей Петри, соответствующего определению**

```
var n, d: INT;
c_code {
    cs_INT v_n, v_d;
}
```

Выражения, описанные на языке CPN ML, транслируются в код на языке С. Для того, чтобы транслировать выражения, транслятор производит обход дерева синтаксического разбора выражения, во время которого, он транслирует подвыражения, соответствующие тем вершинам дерева, все потомки которых уже посещены алгоритмом трансляции. Для хранения промежуточных значений, которые будут получены при вычислении выражений, используются служебные переменные. Трансляция каждого из подвыражений заключается в генерации кода, выполняющего соответствующие семантике подвыражения вычисления и помещающего результат в отведённую переменную. Для обнаружения ошибок, которые могут возникнуть во время вычисления выражений при выполнении модели (например, деление на ноль), при трансляции потенциально ошибочных операций в код, вычисляющий значение выражения, добавляется проверка аргументов операций на допустимые значения конструкцией `assert` языка Promela.

Таким образом, если во время вычисления выражения произойдёт выполнение ошибочной операции, ошибка будет найдена верификатором.

Рассмотрим трансляцию на примере перехода, представленного на рис. 2.

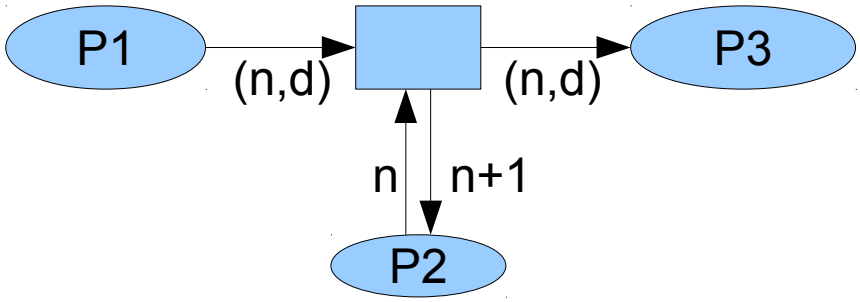


*Рис. 2. Переход РСП*

В качестве типа для мест P1 и P3 выступает кортеж из пары целых чисел

```
IxI = product INT * INT;
```

Место P2 имеет целочисленный тип.

Определение переменных в результирующей модели, соответствующих местам сети Петри:

```
    c_state "msOfIxI p1" "Global"
    c_state "msOfIxI p3" "Global"

//  это место всегда содержит ровно одно значение,
//  поэтому оно представляется не мультимножеством,
//  а переменной типа cs_int
    c_state "cs_int p2" "Global"
```

Код `c_state "<тип данных> <имя переменной>" "Global"` добавляет переменную языка C в вектор состояния Promela-модели. «`Global`» обозначает, что эта переменная является глобальной для модели. Из кода, содержащегося во вставках на языке C, обращение к глобальным переменным модели реализуется с помощью обращений к полям переменной `now`, хранящей во время верификации Promela-модели её текущее состояние [10].

Нахождение значений переменных перехода, при котором он является допустимым, реализуется следующим фрагментом кода на языке Promela:

```
c_code { now.i1 = now.p1.size; } ;
do
    // в цикле осуществляется перебор значений,
    // лежащих в мультимножестве p1
    // значение переменной i1 определяет индекс элемента
    // мультимножества p1, который будет взят
```

```
    // при осуществлении перехода
:: i1 == 0 -> break;
:: else ->
    i1--;
    c_code {
         copy_msOfIxI(&now.p1, &fT1_in1, now.i1);
         // скопировали i1-ый элемент мультимножества p1
         // в переменную fT1_in1
         v_n = fT1_in1.field1; // находим значения
         v_d = fT1_in1.field2; // переменных n и d
         fT1_in2 = now.p2;
         // скопировали элемент мультимножества p2
         // в переменную fT1_in2
// переменные, определяющие возможность осуществления перехода:
         evaluationFail = exprError = false;
         guard = true;
// вычисление выражений, приписанных входным дугам перехода
         // вычисление выражения «(n,d)»
         fT1_in_v1.field1 = v_n;
         fT1_in_v1.field2 = v_d;
         if (cmp_IxI(&fT1_in1, &fT1_in_v1)) {
// если i1-ый элемент мультимножества p1, рассматриваемый
// на текущей итерации, не равен выражению (n,d),
// то при данном значении переменной i1
// переход не является допустимым
              guard = false; goto t1_check;
         }
         fT1_in_v2 = v_n; // вычисление выражения «n»
         if (cmp_int(&fT1_in2, &fT1_in_v2)) {
 // если элемент мультимножества p2 не равен выражению n,
 // то переход не возможен при данном значении переменной i1
              guard = false; goto t1_check;
         }
// вычисление выражений, приписанных выходным дугам перехода
         // вычисление выражения «(n,d)»
         fT1_out1.field1 = v_n;
         fT1_out1.field2 = v_d;
         if (now.p3_.size + 1 > MS_MAX_SIZE) {
// если в выходном мультимножестве нет свободного места,
// то переход не возможен
              evaluationFail = true; goto t1_check;
         }
         // вычисление выражения «n+1»
         fT1_out2 = (v_n + 1);
t1_check:
          if (evaluationFail) guard = false;
    } ;
    if
```

```
    :: c_expr { guard } ->
// если переход возможен при данном значении переменной i1
         trans = 1; // запоминаем номер перехода
         t_i1 = i1; // запоминаем, при каком значении i1
                    // переход возможен
         if // недетерминированно выбираем:
         :: goto doTrans; // либо перейти
                          // к срабатыванию перехода,
         :: skip;     // либо продолжить перебор значений
         fi;
    :: else -> skip;
    fi;
od;
```

Следующий фрагмент кода на языке Promela реализует срабатывание перехода.

```
doTrans:
if
:: trans == 1 -> // осуществляем переход 1
    c_code {
// входные дуги:
         get_msOfIxI(&now.p1, &fT1_in1, now.t_i1);
         v_n = fT1_in1.field1;
         v_d = fT1_in1.field2;
         fT1_in2 = now.p2;
// выходные дуги:
         fT1_out1.field1 = v_n;
         fT1_out1.field2 = v_d;
         put_msOfIxI(&now.p3, &fT1_out1);
         fT1_out2 = (v_n + 1);
         now.p2_ = fT1_out2;
    } ;
// зануляем переменные, которые не должны входить
// в состояние модели
    i1 = 0; t_i1 = 0; trans = 0;
    goto mainLoop; // продолжаем выполнение модели
:: else -> skip; // выполнение модели завершено
fi;
```

### 2.5 Трансляция временных конструкций

Модель, являющаяся результатом трансляции РСП с временными конструкциями, строится по тому же принципу, что и для РСП без таких конструкций, с теми отличиями, что необходимы для моделирования времени. В данном разделе описаны детали трансляции, относящиеся к временным конструкциям.

**Временные метки фишек**

В соответствии с определением времени в РСП, момент времени изменяется только

в том случае, если в состоянии отсутствуют допустимые переходы, поэтому изменять значения временных меток требуется только в конце главного цикла модели и только в случае, если не было найдено допустимых переходов. Чтобы получить выходную модель, соответствующую входной РСП с временными конструкциями, требуется провести трансляцию временных типов, переходов, смежные которым места содержат фишки временных типов, а также изменить условие выхода из главного цикла. Временные типы данных транслируются в структуры с двумя полями, хранящими соответственно значение фишки и временную метку.

**Допустимость переходов**

При проверке допустимости перехода, среди входных мест которого есть места, содержащие фишки временного типа, следует учитывать значения времени, приписанные фишкам так, чтобы при исполнении результирующей модели переход был сочтён недопустимым, если необходимые для осуществления перехода фишки недоступны в соответствующие моменты времени. Это значит, переход должен считаться допустимым, только если в качестве значения выражения на каждой из его входящих дуг может быть выбрана фишка, метка времени которой не превышает значения метки времени, вычисленного в выражении на данной дуге. Если в выражении отсутствует оператор временной задержки, то задержка считается нулевой, то есть фишка, взятая для такой дуги должна быть доступна в текущем состоянии.

**Изменение временных меток фишек**

Если в текущем состоянии модели с временными конструкциями в текущий момент времени ни один переход не является допустимым, то требуется проверить последующие моменты времени на наличие в них допустимых переходов. Для осуществления такой проверки требуется произвести уменьшение меток времени, возможно несколько раз, пока не будет найден допустимый переход, либо все метки времени станут нулевыми. Для этого в конце главного цикла результирующей модели при отсутствии допустимых переходов происходит проверка того, присутствуют ли в разметке мест модели фишки, помеченные положительными значениями времени. Если такие фишки присутствуют, то происходит уменьшение их временные меток на минимальную величину, которая может сделать какой-либо из переходов допустимым. Если же таких фишек нет, то есть все фишки доступны в текущем состоянии, то исполнение модели завершается.

Значение d, на которое следует уменьшить временные метки фишек определяется как ненулевой минимум по всем фишкам разности $T_{token}$ - $T_{place}$, где $T_{token}$ — временная метка фишки, а $T_{place}$ — величина задержки в выражении на выходящей из данного места дуге.

Рассмотрим пример изменения временных меток. Пусть в состоянии модели, показанном на рис. 3 отсутствуют допустимые переходы. Временные метки представлены на рисунке числами, следующими за символом @, а величины, предшествующие этому символу, отображают значения фишек.

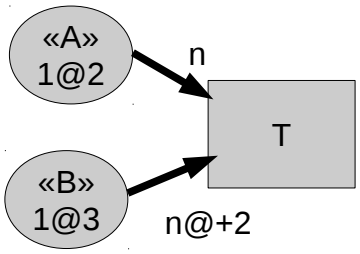


*Рис. 3. Состояние модели без допустимых переходов*

После того как в результате исполнения главного цикла модели было определено, что в текущем состоянии нет допустимых переходов, происходит перебор временных меток фишек. Для места A значение временной метки должно уменьшиться не менее чем на 2, так как дуга выходящая из этого места помечена нулевой задержкой, а фишка, содержащаяся в A помечена значением времени 2. Для места B метку нужно уменьшить на 1. Происходит выбор минимума из этих чисел и значения всех временных фишек уменьшаются на эту величину. Результат этого уменьшения показан на рис. 4.

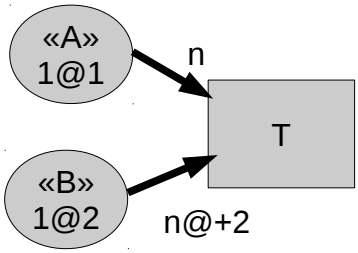


*Рис. 4. Состояние модели после первого уменьшения временных меток*

В данном случае уменьшение временных меток не привело к появлению допустимых переходов. В таком случае уменьшение временных меток будет выполнено ещё раз. Снова выбираем значение, на которое нужно уменьшить метки. Это значение вновь равно 1. Состояние модели, получившееся после второго уменьшения временных меток показано на рис. 5. В этом состоянии переход T становится допустимым при значении 1 переменной n.

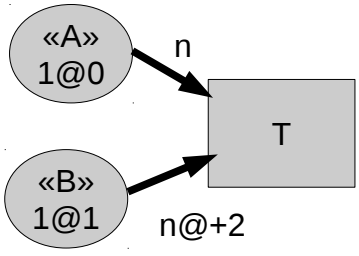


*Рис. 5. Состояние модели после второго уменьшения временных меток*

Теперь, пока в модели будут срабатывать допустимые переходы, значения временных меток фишек, уже находящихся в разметке мест, не будут изменяться; однако в результате срабатывания переходов могут появляться новые временные фишки.

### 2.6 Оптимизация модели на языке Promela

Основной проблемой, возникающей при верификации методом проверки моделей является невозможность сохранить большое пространство состояний в доступном объёме памяти. При большом количестве достижимых состояний осуществление верификации оказывается крайне неэффективным. Оптимизация применяется для обеспечения эффективности результирующей модели, то есть для уменьшения пространства достижимых состояний, а также для уменьшения размера вектора состояния, что позволяет увеличить количество состояний, которые могут храниться в фиксированном объёме памяти. Транслятор использует следующие приёмы оптимизации:

- вставки кода на языке C, скрывающие служебные переменные;
- различное представление значений мультимножеств над различными типами данных.

В некоторых случаях для того, чтобы провести верификацию, требуется исключить некоторые состояния из пространства состояний модели (например, состояния, в которых суммарное количество фишек в некотором множестве мест превышает некоторое заданное число). Для этого можно использовать предикат **`stop_process`**, который записывается на языке C. Этот предикат определяет, следует ли продолжать исполнение модели из данного состояния. Чтобы верификатор рассмотрел все достижимые состояния модели, достаточно определить **`stop_process`** тождественно ложным.

### 2.7 Разработка транслятора

Транслятор, реализующий описанный в данной главе алгоритм, был написан на языке Java. Файл, подающийся на вход транслятора, содержит РСП и имеет формат, совместимый с форматом, используемым системой CPN Tools. В таком файле модель записана в формате XML. Для чтения модели была использована библиотека грамматического анализа XML-документов javax.xml.parsers, входящая в стандартный набор библиотек Java [13]. В процессе чтения входного файла транслятор строит внутреннее представление модели. Внутреннее представление включает в себя список типов данных, список переменных, список мест и список переходов. При построении внутреннего представления иерархия сети разворачивается: все переходы-подстановки заменяются соответствующими уровнями иерархии сети, все места, входящие во

множества-слияния объединяются в одно составное место, таким образом внутреннее представление является неиерархичным.

Построение внутреннего представления включает в себя следующие этапы:

- чтение объявлений типов данных сети Петри, при котором происходит добавление их в список типов данных;
- чтение объявлений переменных сети Петри, при котором происходит добавление их в список переменных;
- чтение уровней иерархии сети Петри, содержащих места, переходы и дуги сети Петри;
- нахождение первичного уровня иерархии; развёртывание иерархии сети Петри, при котором происходит обход в глубину дерева уровней иерархии сети Петри. На этом этапе создаются экземпляры мест и переходов сети Петри и помещаются в соответствующие списки.

Во время построения внутреннего представления происходит синтаксический анализ выражений CPN ML. Во время построения дерева синтаксического разбора происходит проверка типов выражений. Если внутреннее представление модели построено без ошибок, то транслятор производит генерацию кода результирующей модели, выполняя все этапы трансляции.

# ГЛАВА 3. Трансляция автоматных спецификаций в раскрашенные сети Петри

Язык автоматных спецификаций является ёмким и лаконичным. Так как модель на этом языке представляется в виде текстового файла, он удобен для генерации множества похожих моделей, отличающихся друг от друга некоторыми параметрами, например, количеством автоматов в системе. В то же время существуют различные средства анализа и верификации моделей представленных на языке РСП, поэтому представляет интерес трансляция автоматных спецификации в РСП-модели. В этой главе приводится описание работы транслятора из автоматных спецификаций в язык РСП. Для иллюстрации алгоритма трансляции приводится пример трансляции.

## 3.1 Схема трансляции

Алгоритм трансляции и доказательство его корректности приведены в [1]. Трансляция из автоматной спецификации в РСП происходит поэтапно. Сначала транслятор производит чтение входного файла и построение внутреннего представления модели. После этого происходит генерация выходной модели. В это время транслируются переменные и состояния автоматов (в том числе начальные значения), а также переходы. Транслятор поддерживает переменные, имеющие целочисленный и булевский тип данных, а также массивы (в том числе многомерные). На заключительном этапе транслятор записывает сгенерированную модель выходной файл.

Для моделирования недетерминированного выбора одного из константных целочисленных значений поддерживается оператор **`random`**.

Для выходной модели требуется определить следующие типы данных:

- `SIGTYPE` — перечисление типов сигналов из автоматной модели;
- `SIGNAL` — кортеж `SIGTYPE * INT * INT * INT`; переменные такого типа представляют собой сигналы вида (Type, Src, Dest, Param); элементы кортежа обозначают тип сигнала, идентификатор отправителя, идентификатор получателя и параметр сигнала соответственно;
- типы данных для всех встречающихся в модели переменных. Типом, соответствующим массивам является список, локальным переменным — кортеж INT * *тип переменной*. Первым элементом кортежа является идентификатор автомата, которому эта локальная переменная принадлежит, вторым — значение переменной.

Для хранения сигналов окружения в выходной модели создаётся место `ENV`, имеющее тип `SIGNAL`. Для каждого из подвыражений недетерминированного выбора создаётся переменная `nondeterm_#`*`i`*, а также место `NONDETERM_#`*`i`*, разметка которого есть множество возможных альтернатив недетерминированного выбора. При осуществлении такого выбора переменная `nondeterm_#`*`i`* получит одно значение одной из фишек принадлежащих разметки данного места.

Также для результирующей модели определяются целочисленная переменные id, использующаяся для хранения идентификатора автомата при выполнении переходов, а также переменные I_src и I_param целочисленного типа, и переменная I_type, имеющая тип SIGTYPE. Эти переменные используются при извлечении фишки-сигнала из места ENV для хранения его полей.

В качестве примера рассмотрим иллюстративную модель «отправитель-получатель». В модели есть два типа сигналов — сообщение «msg» и подтверждение доставки «ack» Автомат «Sender» моделирует отправитель. Он имеет два состояния: «IDLE», в котором он готов отпавить сообщение получателю и «AWAITING_ACK», в котором он дожидается подтверждения доставки. Его локальная целочисленная переменная «receiver_id» используется для хранения идентификатора автомата-получателя. У автомата два перехода. Переход «send» моделирует отправление сообщения. Он переводит автомат из состояния IDLE в состояние AWAITING_ACK, отправляя сигнал (id, receiver_id, msg, 0). Переход «receive_ack» моделирует получение автоматом подтверждения о доставке сообщения. Данный переход переводит автомат из состояния AWAITING_ACK a IDLE, принимая сигнал, имеющий тип ack. Автомат имеет идентификатор 1, его начальным состоянием является IDLE. Автомат «Receiver» моделирует получатель и имеет единственное состояние «IDLE». Единственный переход «receive» моделирует принятие сообщения и отправку подтверждения о доставке отправителю. Он принимает сигнал I типа msg и отправляет сигнал (id, I.src, ack, 0). Автомат имеет идентификатор 2.

Для данной модели тип SIGTYPE определяется как перечисление, содержащее два элемента «msg» и «ack». Так как в модели используется целочисленная локальная переменная, то в выходной модели определяется тип LOCALINT = product INT * INT.

### 3.2 Трансляция переменных и состояний

Для каждой из переменных, как глобальных, так и локальных (за исключением специальной локальной переменной **`id`**, обозначающий идентификатор автомата) строится

место РСП, имеющее соответствующий тип данных; также строятся соответствующие переменные РСП, предназначенные для доступа к значениям переменных, хранящихся в местах РСП. В случае нашей иллюстративной модели будет построено место Sender__receiver_id, типом которого является LOCALINT, а также переменная с теми же именем и типом.

При трансляции инициализирующей функции для мест, в которые были транслированы переменные, определяется начальная разметка, соответствующая начальным значениям переменных. Значением по умолчанию является 0 для целочисленных переменных и false для булевских. В случае нашей модели в качестве начальной разметки места Sender__receiver_id используется значение (1, 2). Первый элемент кортежа — это идентификатор автомата которому принадлежит локальная переменная, а второй — её начальное значение.

Для каждого из состояний строится место, имеющее тип `INT`. Это место предназначено для хранения фишек, значения которых являются идентификаторами автоматов. Таким образом, при переходе автомата из одного состояния в другое будет перемещена из одного места-состояния в другое фишка, имеющая значение идентификатора автомата. Для нашей модели построены будут три таких места: Sender__IDLE, Sender__ AWAITING_ACK и Receiver_IDLE.

Для каждого из экземпляров автоматов в мультимножество начальной разметки места, соответствующего начальному состоянию автомата, помещается значение идентификатора автомата. В случае нашей модели начальной разметкой мест Sender__IDLE и Receiver_IDLE являются значения 1 и 2 соответственно. Начальная разметка места Sender__AWAITING_ACK пуста.

Начальная разметка места ENV задаётся в соответствии с начальными сигналами среды. Так, как в рассматриваемой модели отсутствуют сигналы, изначально присутствующие в среде, то разметка места ENV пуста.

### 3.3 Трансляция переходов

Для каждого из переходов автоматной спецификации строится переход РСП. Для этого перехода строятся следующие дуги:

- Входящая дуга из места, соответствующего исходному состоянию перехода.
- Исходящая дуга в место, соответствующее результирующему состоянию перехода.
- Входящие и исходящие дуги для всех мест, соответствующих переменным,

используемым в качестве аргументов в Os, P и E данного перехода, а также переменным, стоящим в правой части вычислений E (то есть переменным, значения которых изменяются в результате срабатывания данного перехода). Выражения на дугах при этом строятся следующим образом:

- для глобальных переменных выражение входящей дуги является идентификатором переменной;
- для локальных переменных выражение входящей дуги является кортежем, состоящим из идентификатора автомата и идентификатора переменной;
- для переменных, значения которых данный переход не изменяет, выражения исходящих дуг совпадают с выражениями входящих дуг;
- для переменных, значения которых изменяются при срабатывании данного перехода, выражения исходящих дуг отличаются от выражений входящих дуг заменой идентификатора переменной, на выражение, значение которого ей присваивается;
- для всех выражений недетерминированного выбора входящих в Os, P и E данного перехода добавляются входящие и исходящие дуги в соответствующее место `NONDETERM_#`*`i`*, в качестве выражения для этих дуг выступает идентификатор переменной `nondterm_#`*`i`*.

- Если данный переход срабатывает в результате приёма сообщения, то строится входящая дуга из места `ENV`, выражением на этой дуге является (I_src, id, I_type, I_param).

Предикат срабатывания перехода переводится в охранное выражение.

При трансляции выражений обращения к элементам массивов транслируются в вызов функции **`nth`**, возвращающей элемент списка по индексу. Изменение элемента массива переходом транслируется в вызов функции **`nthreplace`**, возвращающей список, в котором указанный элемент заменён указанным значением.

Рассмотрим, во что будут транслированы переходы иллюстративной модели. Результатом трансляции перехода send автомата Sender является переход РСП Sender__send. Так как транслируемый переход переводит автомат Sender из состояния IDLE в состояние AWAITING_ACK, то инцидентными соответствующему переходу РСП оказываются входящая из места Sender__IDLE и исходящую в место Sender__AWAITING_ACK дуги. Выражением на этих дугах является id. Так как

в результате срабатывания данного перехода происходит отправка сообщения типа msg автомату с идентификатором receiver_id, то в выходной модели присутствует дуга выходящая из перехода Sender__send и ведущая в место ENV. Выражением на этой дуге является (id, Sender__receiver_id, msg, 0). Чтобы при выполнении данного перехода было известно значение переменной Sender__receiver_id, строится двухсторонняя дуга, связывающая переход и место Sender__receiver_id, с именем данной переменной в качестве выражения на дуге.

Переход receive_ack автомата Sender транслируется в переход Sender__receive_ack. Также как и переходу рассмотренному выше, данному переходу инцидентны две дуги с выражением id, связывающие места, соответствующие исходному и результирующему состояниям автомата. Так как при выполнении данного перехода выполняется приём сообщения, то для него строится входящая из места ENV дуга. Условие срабатывания перехода транслируется в охранное выражение «I_type = ack».

На рис. 6 приведена РСП, являющаяся результатом трансляции иллюстративной модели.

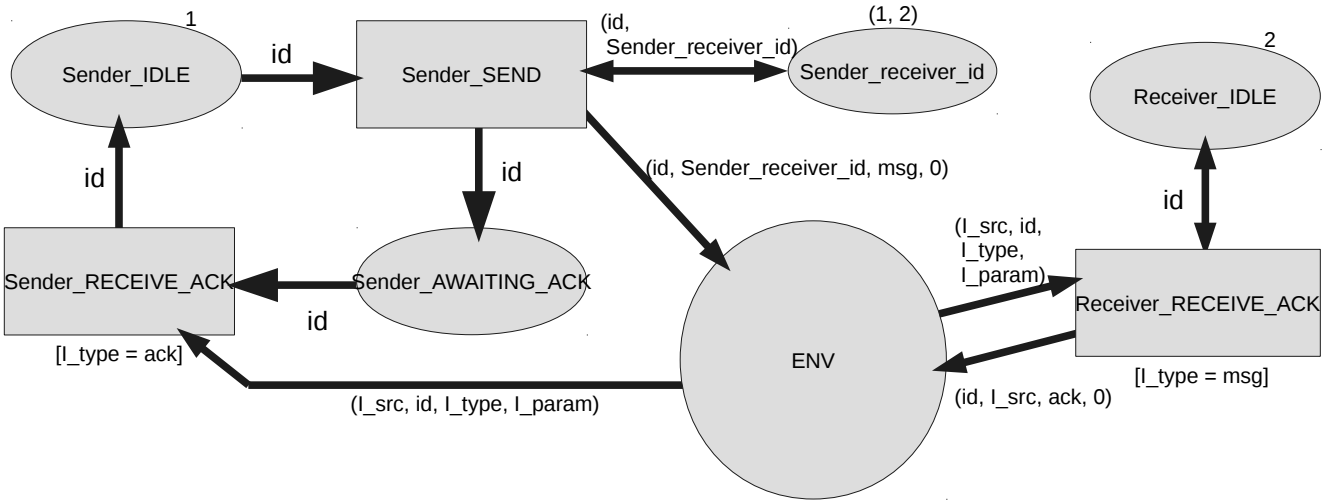


*Рис. 6. Результирующая РСП-модель*

### 3.4 Разработка транслятора

Транслятор из автоматных спецификаций в РСП был разработан на языке Java. На вход транслятору подаётся файл, содержащий модель на языке автоматных спецификаций. Грамматика используемого языка приведена в приложении А. Данная грамматика принадлежит классу грамматик LL(1), а значит может быть разобрана грамматический анализатором, реализующим алгоритм рекурсивного спуска [14]. Такой анализатор был разработан и использован для чтения входных файлов транслятором. В процессе чтения файла транслятор строит внутреннее представление модели. Внутреннее представление системы взаимодействующих автоматов включает в себя

список типов сигналов, список переменных, список классов автоматов, список экземпляров автоматов, множество начальных сигналов, а также начальные значения переменных. Построение внутреннего представления модели содержащейся во входном файле заключается в синтаксическом разборе этого файла транслятором. На этапе синтаксического разбора транслятор производит проверку типов выражений, используемых в модели. Если входной файл прочитан, и внутреннее представление модели построено успешно, то транслятор выполняет над ним этапы алгоритмы трансляции, описанные в данной главе. Результатом выполнения трансляции является внутреннее представление РСП-модели, включающее в себя список типов, список переменных, список мест и список переходов. Далее внутреннее представление получившейся РСП записывается в выходной файл в формате XML, совместимом с форматом, который используется системой CPN Tools.

# ГЛАВА 4. Верификация коммуникационных протоколов

Для верификации РСП-моделей протоколов использовался транслятор, представленный в главе 3 и система верификации SPIN, как показано на схеме, изображёной на рис. 7.

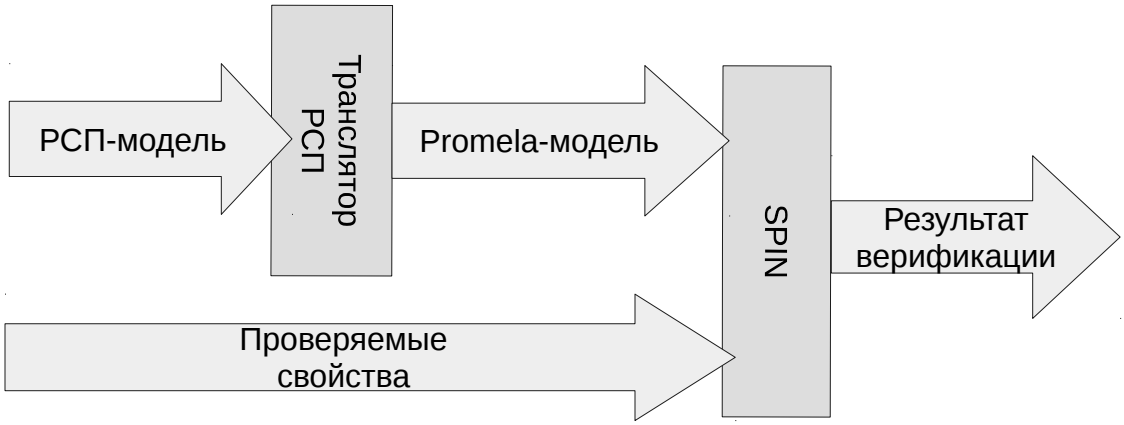


*Рис. 7. Схема системы верификации РСП*

Чтобы верифицировать автоматные спецификации, помимо прочих компонентов системы верификации использовался транслятор, представленный в главе 2, что изображено на схеме, изображёной на рис. 8.

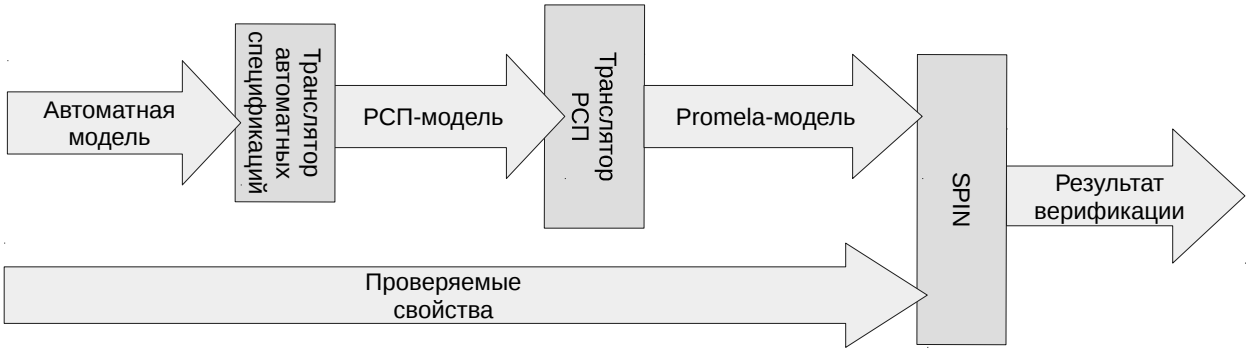


*Рис. 8. Схема системы верификации автоматных спецификаций*

## 4.1 UCM-специфицированный протокол

Данный протокол был изначально описан на языке UCM, а затем транслирован в РСП [15]. Протокол содержит отправитель и получатель, передающие данные в ненадёжной среде. Для верификации протокола использовалась часть представляемой системы — транслятор из РСП в Promela.

При верификации проверялось следующее свойство: всегда последовательность принятых получателем пакетов является префиксом последовательности, которую передаёт отправитель. Первая версия протокола содержала ошибку, при которой получатель принимал один высланный пакет данных несколько раз. Верификация показала, что свойство не выполняется, был найден контрпример для Promela-модели,

по которому был построен контрпример для модели представленной в виде РСП.

Вторая версия этого протокола исключала ситуацию, в которой возникала ошибка. Для неё была также была проведена верификация, проверяющая то же свойство. Верификация показала, что для второй версии протокола свойство выполняется. Сведения о работе верификатора для данной модели приведены в таблице 1.

Сведения о работе верификатора Таблица 1

| | Модель первой версии протокола (верификатор прекратил работу после найденной ошибки) | Модель второй версии протокола |
|---|---|---|
| Число посещённых состояний модели | 23689479 | 14715011 |
| Число выполненных переходов | 88134826 | 53521792 |
| Объём использованной памяти | 2660 мегабайт | 3956 мегабайт |
| Время работы | 583 секунды | 388 секунд |

### 4.2 SDL-специфицированный протокол PAR

Данный протокол описан в книге [16]. Модель содержит отправитель и получатель в ненадёжной среде. Отправитель передаёт получателю определённую последовательность пакетов в определённом порядке один за другим. Получатель отправляет подтверждение о получении пакета. Отправитель использует таймер для повторной отправки пакета, если не получает подтверждение. Для моделирования таймера используется понятие времени в РСП. В протоколе используется бит чётности номера пакета для проверки того, что полученный пакет не является повторно принятым.

Была проведена верификация этого протокола, проверялись следующие свойства:

1. Всегда последовательность принятых пакетов является префиксом последовательности отправляемых. Свойство представлено в виде формулы LTL

□ `received_prefix`

Здесь <`received_prefix`> — это следующий предикат: последовательность принятых пакетов является префиксом отправляемой последовательности. Так как в данной модели последовательность отправляемых пакетов — это три целых значения 5, 6, 7, то для РСП-модели этот предикат определяется как

```
let
 val received = ParProtocolmainpage.Network.
                MyReceiver.Receiver_User
in
 ((length received) < 1)
 orelse
```

```
 (#integer2 (nth 0 received) = 5)
       andalso ((length received) < 2)
 orelse
 (#integer2 (nth 1 received) = 6)
       andalso ((length received) < 3)
 orelse
 (#integer2 (nth 2 received) = 7)
       andalso ((length received) < 4)
end
```

2. Всегда если отправитель начал отправлять пакет с определённым содержимым, то если в дальнейшем какой-либо пакет будет принят отправителем, то отправителем будет принят пакет с таким содержимым. Представляющая это свойство формула LTL записывается как

□ ( sent_c → ((◊ received) → (◊ received_c)) )

Здесь предикат <sent_c> значит, что выполнен переход РСП, соответсвующий началу отправки пакета с некоторым содержимым (например, содержимым пакета будет число 5). Предикат <received> значит, что был выполнен переход, соответствующий приёму пакета, а <received_c> - что был выполнен этот переход и содержимое принятого пакета совпадает с ожидаемым. Для РСП эти предикаты определяются так:

```
sent_c:
ParProtocolmainpage.Network.MySender.MWaitRequestSEnv.
 MBegin2._occurs
andalso ParProtocolmainpage.Network.MySender.MWaitRequestSEnv.
 MBegin2.M_integer1val = 5

received:
ParProtocolmainpage.Network.MyReceiver.MyWaitDataframe.
 Myoutput_SEnvOut_Receiver_User._occurs

received_c:
ParProtocolmainpage.Network.MyReceiver.MyWaitDataframe.
 Myoutput_SEnvOut_Receiver_User._occurs
andalso
ParProtocolmainpage.Network.MyReceiver.MyWaitDataframe.
 Myoutput_SEnvOut_Receiver_User.My_fr_data = 5
```

В данном контексте истинность <`_occurs`> для перехода обозначает, что в текущее состояние модель была переведена в результате срабатывания данного перехода.

3. В конце выполнения модели, то есть в том состоянии, в котором отсутствуют допустимые переходы, количество принятых пакетов равно количеству отправленных. Для проверки этого свойства был переопределён предикат (функция языка C) <`valid_end_state`>, возвращающий истинное значение в том и только в том случае, если длина последовательности принятых пакетов совпадает с длиной отправляемой последовательности. В конечном состоянии Promela-модели значение этого предиката проверяется с помощью утверждения (assert).

Верификация с помощью системы SPIN показала, что данные свойства для модели выполняются. В таблице 2 приведены сведения о работе верификатора для данной модели.

Сведения о работе верификатора для модели протокола PAR Таблица 2

| | |
|---|---|
| Число посещённых состояний модели | 410 |
| Число выполненных переходов | 595 |
| Объём использованной памяти | 136 мегабайт |
| Время работы | 0.04 секунды |

### 4.3 Кольцевой RE-протокол

RE-протокол применяется в кольцевых сетях, в которых передача данных производится всеми станциями синхронно в регулярные моменты [1, 17]. От станции к станции по кольцу передаётся фрейм фиксированной длины. Во время функционирования кольцевой сети каждая станция передает полученный фрейм своему ближайшему соседу по кольцу. Станция, которая собирается отправить данные, ждет, когда она получит фрейм, помеченный как «пустой», меняет его метку на «занятый», и помещает свои данные в полученный фрейм. Станция, получившая содержащий данные, то есть «занятый» фрейм, передает его неизменным, если адрес назначения данных отличен от её адреса. Если же станция получает фрейм, содержащий адресованные ей данные, то она копирует данные фрейма в свой буфер, но не изменяет фрейм. Кроме станций в кольце присутствует, так называемый *монитор*, функцией которого является контроль над работой протокола. При моделировании протокола для нас представляют интерес следующие поля фрейма:

- R и E – два бита метки;
- DEST – адрес станции-получателя данных;
- SRC – адрес станции-отправителя данных.

Значения битов R и E несут следующий смысл:

- R = 0, E = 0 – пустой фрейм;
- R = 0, E = 1 – фрейм, содержащий данные;
- R = 1, E = 0 – сигнал монитору, означающий, что требуется произвести реинициализацию кольца из-за ошибки, обнаруженной станцией;
- R = 1, E = 1 – реинициализация станций в кольце, производящаяся по инициативе монитора

При отправке данных станция переходит из состояния LISTEN в состояние ACTIVE и изменяет метку на 01. При возвращении фрейма такой станции, она в случае, если метка имеет значение 01, а поле SRC – её адрес, переходит в состояние LISTEN изменяет метку на 00. В противном случае она переходит в состояние RECOVERY, в котором она изменяет все значения метки, кроме 11, на 10, что позволяет оповестить монитор о произошедшей ошибке. Получая в состоянии RECOVERY значение метки 11, станция переходит в состояние LISTEN, оставляя фрейм неизменным. Станция, получившая метку 10, переходит в состояние RECOVERY, независимо от своего текущего состояния, метка 11 таким же образом вызывает переход в состояние LISTEN.

Монитор находится в состоянии NORMAL до тех пор, пока он считает, что кольцо функционирует нормально. Если он сам обнаруживает ошибку, он переходит в состояние RESTORE. Если он получает метку 10, сигнализирующую о том, что необходимо реинициализировать кольцо, он переходит в состояние RESET.

### *4.3.1 Автоматная модель RE-протокола*

**Модель станции**

Множество состояний автомата, моделирующего станцию, S = {LISTEN, ACTIVE, RECOVERY}. Автомат имеет одну локальную переменную n_id – идентификатор следующей станции в кольце. Переходы автомата приведены в таблице 3.

Переходы модели станции Таблица 3

| № | Ss | Se | P | Os |
|---|---|---|---|---|
| 1. | LISTEN | LISTEN | I.Type = 11 | (id, n_id, 11, I.Value) |
| 2. | LISTEN | LISTEN | I.Type = 01 | (id, n_id, 01, I.Value) |
| 3. | LISTEN | LISTEN | I.Type = 00 | (id, n_id, 00, I.Value) |
| 4. | LISTEN | ACTIVE | I.Type = 00 | (id, n_id, 01, (id, Rnd x)) |
| 5. | ACTIVE | LISTEN | I.Type = 01 && I.Value.Src = id | (id, n_id, 00, (null, I.Value.Dest)) |
| 6. | ACTIVE | LISTEN | I.Type = 11 | (id, n_id, 11, I.Value) |
| 7. | ACTIVE | RECOVERY | I.Type = 00 \|\| (I.Type = 01 && I.Value.Src != id) | (id, n_id, 00, (null, I.Value.Dest) |
| 8. | ACTIVE | RECOVERY | I.Type = 10 | (id, n_id, 10, I.Value) |
| 9. | LISTEN | RECOVERY | I.Type = 10 | (id, n_id, 10, I.Value) |
| 10. | RECOVERY | LISTEN | I.Type = 11 | (id, n_id, 11, I.Value) |
| 11. | RECOVERY | RECOVERY | I.Type = 00 \|\| I.Type = 01 \|\| I.Type = 10 | (id, n_id, 10, I.Value) |

**Модель монитора**

Множество состояний автомата, моделирующего монитор, S = {NORMAL, RESET, RESTORE}. Автомат имеет две локальные переменные: n_id – идентификатор следующей станции в кольце и last_src. Последняя переменная используется для дополнительного контроля за работой кольца. Переходы автомата представлены в таблице 4.

Переходы модели монитора Таблица 4

| № | Ss | Se | P | Os | E |
|---|---|---|---|---|---|
| 1. | NORMAL | NORMAL | I.Type = 00 | (id, n_id, 00, I.Value) | last_src = 0 |
| 2. | NORMAL | NORMAL | I.Type = 01 && I.Value.Src != 0 && I.Value.Src != last_src | (id, n_id, 01, I.Value) | last_src = I.Value.Src |
| 3. | RESET | NORMAL | I.Type = 11 | (id, n_id, 00, I.Value) | last_src = 0 |
| 4. | NORMAL | RESET | I.Type = 10 \|\| I.Type = 11 | (id, n_id, 11, I.Value) | |
| 5. | RESET | RESET | I.Type = 00 \|\| I.Type = 01 \|\| I.Type = 10 | (id, n_id, 11, I.Value) | |
| 6. | RESTORE | RESET | I.Type = 10 \|\| I.Type = 11 | (id, n_id, 11, I.Value) | |
| 7. | RESTORE | NORMAL | I.Type = 00 \|\| I.Type = 01 | (id, n_id, 00, I.Value) | last_src = 0 |
| 8. | NORMAL | RESTORE | I.Type = 01 & (I.Value.Src = 0 \|\| I.Value.Src = last_src) | (id, n_id, 01, I.Value) | |

**Взаимодействие автоматов**

Взаимодействие автоматов организовано следующим образом: станции пронумерованы от *1* до *N*, монитору присвоен номер *N+1*, и для каждого автомата с номером *i* его *n_id = i+1*. Исходный сигнал равен (*N+1, 1, 00, (0,0)*). Переменной монитора *last_src* присвоено исходное значение *0*.

### *4.3.2 Верификация автоматной модели RE-протокола*

RE-протокол был верифицирован для случая когда работа происходит в надёжной среде, а количество станций варьируется от двух до десяти. Проверяемые свойства:

- отправленные пакеты будут обязательно приняты;
- отсутствие тупиков.

Для проверки того, что тупиковых состояний нет в конечном состоянии Promela-модели было добавлено утверждение assert(false). Для проверки того, что отправленные пакеты будут приняты, в вектор состояния модели были добавлены переменные-счётчики. При отправке сообщения счётчик инкрементировался, при принятии – декрементировался. LTL-формула, выражавшая свойство имела следующий вид: «□ ◊ (counter[i]=0)».

Верификация показала, что проверяемые свойства выполняются.

## 4.4 Кольцевой ATMR-протокол

ATMR-протокол является одним из стандартов ISO для кольцевых сетей с высокоскоростной передающей средой. В отличие от RE-протокола, в ATMR-протоколе по кольцу может перемещаться не один кадр, а несколько ячеек. Ячейка может быть одного их следующих типов: Empty, Data и Reset. Ячейка, имеющая тип Empty, является пустой. Станция, получившая пустую ячейку, может использовать её для передачи данных. Ячейка, в которую помещены данные, имеет тип Data. Станция, получающая Data-ячейку, в случае, если данные, содержащиеся в ячейке, адресованы ей, копирует их в локальный буфер и изменяет её тип на Empty, то есть освобождает ячейку. Другие станции при получении Data-ячейки пропускают её дальше по кольцу. Изначально каждая из станций имеет возможность отправить ограниченное количество сообщений. Это количество называется *числом кредитов*. Каждый раз при отправке сообщения станцией количество её кредитов уменьшается на 1. Число кредитов отправлявшей сообщения станции в какой-то момент оказывается равно 0, и станция не сможет отправлять сообщения. Reset-ячейка позволяет возобновить количество кредитов: при получении ячейки такого типа, станция сбрасывает значение числа кредитов, делая его максимальным. Поле Busy Address ячейки, в котором хранится адрес последней станции, отправлявшей данные, предназначено для контроля за работой кольца. Будем моделировать работу ATMR-протокола с одной ячейкой.

### *4.4.1 Автоматная модель ATMR-протокола*

Множество состояний автомата станции S = {IDLE, SEND, RESET, WAIT}. Тип ячейки смоделируем с помощью типа сигнала. Для этого определим множество типов сигналов как {E, D, R}. Поля фрейма DST и BA можно хранить в глобальных переменных, так как количество фреймов в модели не изменяется.

Автомат станции имеет следующие локальные переменные:

- n_id – идентификатор следующей станции в кольце;
- C – текущее число кредитов;

- MaxCr – максимальное число кредитов для данной станции.

Переходы автомата представлены в таблице 5.

Модель станции Таблица 5

| № | Ss | Se | P | Os | E |
|---|---|---|---|---|---|
| 1. | IDLE | IDLE | I.Type = D && I.Value.DST = id | (id, n_id, E, (0, I.Value.BA)) | |
| 2. | IDLE | IDLE | I.Type = D && I.Value.DST != id | (id, n_id, D, I.Value) | |
| 3. | IDLE | IDLE | I.Type = R | (id, n_id, R, I.Value) | C = MaxCr |
| 4. | IDLE | IDLE | I.Type = E && I.Value.BA != id | (id, n_id, E, I.Value) | |
| 5. | IDLE | SEND | Has_data && C > 0 | | |
| 6. | IDLE | RESET | I.Type = E && I.Value.BA = id | (id, n_id, R, I.Value) | |
| 7. | IDLE | WAIT | Has_data && C <= 0 | | |
| 8. | SEND | IDLE | I.Type = E | (id, n_id, D, (random x, id)) | C = C - 1 |
| 9. | SEND | IDLE | I.Type = R | (id, n_id, R, I.Value) | C = MaxCr |
| 10. | SEND | SEND | I.Type = D && I.Value.DST = id | (id, n_id, E, (0, id)) | |
| 11. | SEND | SEND | I.Type = D && I.Value.DST != id | (id, n_id, D, (I.Value.DST, id)) | |
| 12. | RESET | IDLE | I.Type = R | (id, n_id, D, (I.Value.DST, id)) | C = MaxCr |
| 13. | RESET | RESET | I.Type = D && I.Value.DST = id | (id, n_id, E, (0, I.Value.BA)) | |
| 14. | RESET | RESET | I.Type = D && I.Value.DST != id | (id, n_id, D, I.Value) | |
| 15. | RESET | RESET | I.Type = E | (id, n_id, E, I.Value) | |
| 16. | WAIT | IDLE | I.Type = R | (id, n_id, R, I.Value) | C = MaxCr |
| 17. | WAIT | RESET | I.Type = E && I.Value.BA = id | (id, n_id, R, I.Value) | |
| 18. | WAIT | WAIT | I.Type = D && I.Value.DST = id | (id, n_id, E, (0, I.Value.BA)) | |
| 19. | WAIT | WAIT | I.Type = D && I.Value.DST != id | (id, n_id, D, I.Value) | |
| 20. | WAIT | WAIT | I.Type = E && I.Value.BA != id | (id, n_id, E, I.Value) | |

### *4.4.2 Верификация автоматной модели ATMR-протокола*

ATMR-протокол был верифицирован в надёжной среде с количество станций от двух до четырёх и числом кредитов от двух до пяти. Проверяемые свойства:

- отсутствие тупиков;
- все отправленные сообщения будут приняты;
- число принятых сообщений не превосходит числа отправленных.

Для проверки того, что тупиковых состояний нет в конечном состоянии Promela-модели было добавлено утверждение assert(false). Для проверки того, что отправленные пакеты будут приняты, в вектор состояния модели была добавлена переменная-счётчик. При отправке сообщения первой станции счётчик инкрементировался, при принятии ею сообщения – декрементировался. Первая станция была выбрана в силу симметрии протокола — все станции в кольце одинаковы, и если свойство выполняется для одной из них, то можно утверждать, что оно будет выполняться и для остальных. LTL-формула, выражавшая свойство имела следующий вид:

```
( □ ◊ counter==0 ) & ( □ !(counter < 0) )
```

Верификация показала, что проверяемые свойства выполняются.

### 4.5 Телефонные сети с дополнительными функциональностями

Система верификации была применена для исследования проблемы взаимодействия функциональностей в телефонных сетях. Данная проблема заключается в том, что при наличии нескольких различных функциональностей в таких системах, как телефонные сети, может возникать нежелательные отклонения в поведении системы, вызванные взаимодействием функциональностей друг с другом [1, 18]. Были рассмотрены следующие функциональности в телефонных сетях:

- переадресация звонков на заданный номер, если номер занят (call forwarding when busy, CFB);
- безусловная переадресация звонков на заданный номер (call forwarding unconditional, CFU);
- прямое соединение — сразу после того как трубка снята совершается вызов на заданный номер (direct connect, DC);
- запрет исходящих звонков на заданный номер (outgoing dial screening, ODS);
- запрет входящих звонков с заданного номера (terminating call screening, TCS);
- запрет всех входящих звонков (denied termination, DT).

#### *4.5.1 Автоматная модель телефонной сети*

Каждый из телефонов соединён со станцией отдельной линией (для телофона i логически обозначим линию как $env_i$). Каждая из $env_i$ содержит не более одного сигнала. Для того, чтобы поддерживать это свойство используем глобальный булевский массив free. Сигнал по линии может посылать данный телефон либо станция (автоматы BCS или FM), только если эта линия свободна, то есть предыдущий посланный сигнал обработан. Схема модели приведена на рис. 9.

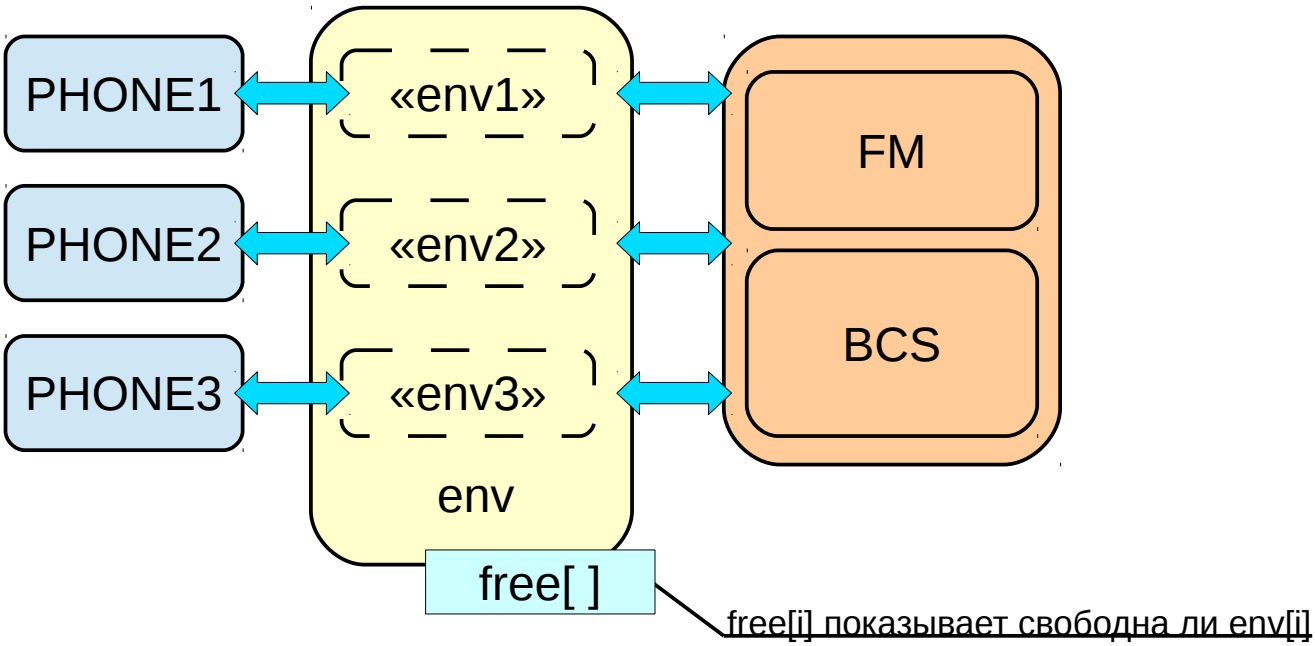


*Рис. 9. Схема модели телефонной сети*

Пусть PHONE_N — количество телефонов в модели.

**Определения**

Глобальные массивы:

bool busy[ PHONE_N ] — показывает занят ли данный номер (начальное значение false);

bool free[ PHONE_N ] — означает, что канал передачи данных свободен (начальное значение true);

int caller[ PHONE_N ] — номер звонящего абонента (начальное значение NO_PHONE);

int callee[ PHONE_N ] - номер абонента, которому звонит данный абонент (начальное значение NO_PHONE).

Глобальные массивы, содержащие сведения о подключенных абонентами функциональностях:

int dc[ PHONE_N ];

int cfb[ PHONE_N ];

int cfu[ PHONE_N ];

int ods[ PHONE_N ];

int tcs[ PHONE_N ];

bool dt[ PHONE_N ].

Глабальная переменная int fm_id — идентификатор автомата, моделирующего менеджер функциональностей.

Множество типов сигналов:

onhook — трубка повешена;

offhook — трубка снята;

dial — набор номера;

dialtone — непрерывный гудок;

busytone — короткие гудки;

callingtone — длинные гудки;

incoming_call — входящий звонок;

call_accepted — вызов принят;

call_dropped — вызов отклонён;

connect — соединение со звонящим при снятии трубки.

**Автомат PHONE (телефон)**

Множество состояний:

IDLE — ничего не происходит, трубка повешена;

PHONE_RINGS — телефон звонит;

OFFHOOK — трубка снята;

DIALTONE — идёт непрерывный гудок;

DIALLING — набран номер, ожидаются гудки;

BUSYTONE — идут короткие гудки;

CALLING — идут длинные гудки;

TALKING — происходит разговор.

Локальный массив

int other[ PHONE_N - 1 ] — номера всех других телефонов.

Множество переходов автомата, моделирующего телефон, приводится в таблице 6.

Переходы телефона Таблица 6

| № | Ss | Se | Os | P | E |
|---|---|---|---|---|---|
| 1 | IDLE | OFFHOOK | (id, fm_id, offhook) | free[ id ] | free[ id ] = false |
| 2 | IDLE | PHONE_RINGS | | I.type == incoming_call | free[ id ] = true |
| 3 | OFFHOOK | DIALTONE | | I.type == dialtone | free[ id ] = true |
| 4 | DIALTONE | IDLE | (id, fm_id, onhook) | free[ id ] | free[ id ] = false |
| 5 | DIALTONE | DIALLING | (id, fm_id, dial, other[ random(PHONE_N - 1) ]) | free[ id ] | free[ id ] = false |
| 6 | DIALLING | BUSYTONE | | I.type == busytone | free[ id ] = true |
| 7 | DIALLING | CALLING | | I.type == callingtone | free[ id ] = true |
| 8 | BUSYTONE | IDLE | (id, fm_id, onhook) | free[ id ] | free[ id ] = false |
| 9 | CALLING | IDLE | (id, fm_id, onhook) | free[ id ] | free[ id ] = false |
| 10 | CALLING | TALKING | | I.type == call_accepted | free[ id ] = true |
| 11 | TALKING | BUSYTONE | | I.type == call_dropped | free[ id ] = true |
| 12 | TALKING | IDLE | (id, fm_id, onhook) | free[ id ] | free[ id ] = false |
| 13 | PHONE_RINGS | OFFHOOK | (id, fm_id, offhook) | free[ id ] | free[ id ] = false |
| 14 | PHONE_RINGS | IDLE | (id, fm_id, onhook) | I.type == call_dropped | |
| 15 | OFFHOOK | TALKING | | I.type == connect | free[ id ] = true |
| 16 | OFFHOOK | CALLING | | I.type == callingtone | free[ id ] = true |
| 17 | OFFHOOK | BUSYTONE | | I.type == busytone | free[ id ] = true |

**Автомат BCS (телефонная станция)**

Данный автомат имеет одно состояние IDLE. Множество переходов автомата приводится в таблице 7.

Переходы телефонной станции Таблица 7

| № | Os | P | E |
|---|---|---|---|
| 1 | (id I.src dialtone) | I.type == offhook &&<br>caller[ I.src ] == NO_PHONE | busy[ I.src ] = true |
| 2 | | I.type == onhook &&<br>caller[ I.src ] == NO_PHONE &&<br>callee[ I.src ] == NO_PHONE | busy[ I.src ] = false<br>free[ I.src ] = true |
| 3 | | I.type == onhook &&<br>callee[ I.src ] != NO_PHONE &&<br>! free[ callee[ I.src ] ] | busy[ I.src ] = false<br>free[ I.src ] = true<br>caller[ callee[ I.src ] ] =<br>NO_PHONE<br>callee[ I.src ] = NO_PHONE |
| 4 | (id, callee[ I.src ],<br>call_dropped) | I.type == onhook &&<br>callee[ I.src ] != NO_PHONE &&<br>free[ callee[ I.src ] ] | busy[ I.src ] = false<br>free[ I.src ] = true<br>free[ callee[ I.src ] ] = false<br>caller[ callee[ I.src ] ] =<br>NO_PHONE<br>callee[ I.src ] = NO_PHONE |
| 5 | (id, caller[ I.src ],<br>call_dropped) | I.type == onhook &&<br>caller[ I.src ] != NO_PHONE &&<br>free[ caller[ I.src ] ] | busy[ I.src ] = false<br>free[ I.src ] = true<br>free[ caller[ I.src ] ] = false<br>callee[ caller[ I.src ] ] =<br>NO_PHONE<br>caller[ I.src ] = NO_PHONE |
| 6 | (id, I.src, busytone) | I.type == dial &&<br>busy[ I.param ] | |
| 7 | (id, I.src, callingtone)<br><br>(id, I.param, incoming_call) | I.type == dial &&<br>! busy[ I.param ] &&<br>free[ I.param ] | busy[ I.src ] = true<br>busy[ I.param ] = true<br>free[ I.param ] = false<br>caller[ I.param ] = I.src<br>callee[ I.src ] = I.param |
| 8 | (id, I.src, connect)<br><br>(id, caller[ I.src ],<br>call_accepted) | I.type == offhook &&<br>caller[ I.src ] != NO_PHONE &&<br>free[ caller[ I.src ] ] | busy[ I.src ] = true<br>free[ caller[ I.src ] ] = false |

**Автомат FM (менеджер функциональностей)**

Данный автомат имеет единственное состояние IDLE. Его локальная переменная bcs_id — идентификатор автомата BCS. Множество переходов автомата приводится в таблице 8.

Переходы менеджера функциональностей Таблица 8

| № | Os | P | E |
|---|---|---|---|
| 1 | (I.src, id, dial, dc[ I.src ]) | I.type == offhook &&<br>dc[ I.src ] != NO_PHONE &&<br>caller[ I.src ] == NO_PHONE | busy[ I.src ] = true |
| 2 | (id, I.src, busytone) | I.type == dial &&<br>dt[ I.param ] | |
| 3 | (I.src, id, dial, cfb[ I.param ]) | I.type == dial &&<br>cfb[ I.param ] != NO_PHONE &&<br>busy[ I.param ] | |
| 4 | (I.src, id, dial, cfu[ I.param ]) | I.type == dial &&<br>cfu[ I.param ] != NO_PHONE | |
| 5 | (id, I.src, busytone) | I.type == dial &&<br>I.param == ods[ I.src ]&&<br>ods[ I.src ] != NO_PHONE | |
| 6 | (id, I.src, busytone) | I.type == dial &&<br>I.src == tcs[ I.param ] | |
| 7 | (I.src, bcs_id, I.type, I.param) | Не выполнены условия остальных переходов | |

### *4.5.2 Верификация автоматных моделей телефонных сетей*

Была проведена верификация ряда моделей, представляющих различные комбинации указанных функциональностей в сети с тремя и четырьмя телефонами. В каждой из моделей присутствовало две функциональности с различными параметрами, подключенные у разных абонентов. Модели изначально были представлены на языке взаимодействующих конечных автоматов. Телефонная станция, обеспечивающая обработку основной функциональности телефонной сети, и каждый из телефонов были представлены в модели отдельным конечным автоматом, кроме того ещё один автомат в модели, так называемый «менеджер функциональностей», необходим для обеспечения обработки дополнительных функциональностей. Для того, чтобы верифицировать модели, представленые на языке автоматных спецификаций, такие модели были транслированы в РСП, после чего была верифицирована полученная модель.

**Проверяемые свойства:**

1. Отсутствие тупиков.

Для проверки данного свойства был использован тот же приём, который был применён для проверки аналогичного свойчтва ATMR-протокола: предикат <valid_end_state> был переопределён, чтобы всегда возвращать ложное значение. Истинность этого предиката проверяется результирующей моделью в состоянии, из которого нет предиката. Таким образом, если в модели присутствует тупик, то будет найдена ошибка.

Это свойство выполняется для всех проверяемых моделей.

2. Свойство безопасности, выраженное LTL-формулой «□ valid_state», где предикат <valid_state> определён на состояниях модели следующим образом — он возвращает false, если хотя бы одно из следующих условий выполнено хотя бы для одного из абонентов:

- у абонента подключена CFU, и у него звонит телефон;
- у абонента подключена DC, и он слышит непрерывный гудок;
- у абонента подключена ODS, он набрал номер, на который ему запрещён звонок и слышит длинные гудки;
- у абонента подключена TCS, и у него звонит телефон, при этом ему звонят с номера, который запрещён;
- у абонента подключена DT, и у него звонит телефон;

В ином случае предикат возвращает true.

Данное свойство выполняется для всех проверяемых моделей.

3. Отсутствие зацикливаний менеджера функциональностей. Данное свойство выражается LTL-формулой «□ ◊ fm_t», где <fm_t> — булева переменная, которая была добавлена в исходную автоматную модель для проверки данного свойства. Смысл значения этой переменной — является ли последний сработавший переход переходом менеджера функциональностей.

Данное свойство выполняется для всех проверяемых моделей, за исключением тех, в которых присутствуют комбинации функциональностей CFB+CFB, CFB+CFU, CBU+CFU. Проблема возникает, если в модели присутствует циклическая переадресация.

После того как было обнаружено, что в модели присутствует ошибка, были внесены исправления, необходимые для устранения этой ошибки. Изменения заключались в следующем: если менеджер функциональностей обнаруживает, что звонок некоторого абонента переадресуется этому же абоненту, то он передаёт данному абоненту сигнал «занято». Была проведена верификация исправленных моделей с теми же свойствами, причём ошибок не обнаружено.

# ЗАКЛЮЧЕНИЕ

В рамках данной работы была разработана и реализована система верификации автоматных спецификаций, использующая раскрашенные сети Петри для промежуточного представления моделей и позволяющая проверять свойства, описанные линейной темпоральной логикой. Для этого были созданы транслятор автоматных спецификаций в раскрашенные сети Петри и транслятор из раскрашенных сетей Петри в язык Promela, являющийся входным для системы SPIN. Были проведены эксперименты по верификации ряда моделей распределённых систем, представленных в виде систем взаимодействующих автоматов и раскрашенных сетей Петри.

Была верифицирована модель телекоммуникационного протокола, специфицированная на языке UCM. Одна из версий этого протокола содержала ошибку, которая была обнаружена верификатором. Был проведен успешный эксперимент по верификации модели SDL-специфцированного протокола PAR, использующей временные конструкции. Были верифицированы автоматные модели протоколов RE и ATMR, предназначенных для использования в кольцевых телекоммуникационных сетях. Также были проведены эксперименты по верификации моделей телефонных сетей с дополнительными функциональностями. По сравнению с экспериментами, описанными в [1], были промоделированы три типа функциональностей. В результате верификации было найдено нежелательное взаимодействие функциональностей, которое затем было устранено. Проведенные эксперименты продемонстрировали работоспособность созданной системы.

Результаты данной работы были представлены на 50-ой и 51-ой Международных научных студенческих конференциях «Студент и научно-технический прогресс» [19, 20] и на международном семинаре «The Fourth Workshop on Program Semantics, Specification and Verification: Theory and Application (PSSV 2013)» [9].

# ЛИТЕРАТУРА


1. **Белоглазов Д.М., Машуков М.Ю., Непомнящий В.А.** Верификация телекоммуникационных систем, специфицированных взаимодействующими конечными автоматами, с помощью раскрашенных сетей Петри // Моделирование и анализ информационных систем, том 18, номер 4, 2011, с. 144-156

2. **Jensen K., Kristensen, L. M.** Coloured Petri nets: modelling and validation of concurrent systems, Springer, 2009

3. **Billington J., Gallasch G. E., Han. B.** A Coloured Petri Net Approach to Protocol Verification // ACPN 2003, LNCS 3098, pp. 210-290

4. **Kristensen L. M., Simonsen K. I. F.** Application of Coloured Petri Nets for Functional Validation of Protocols Designs // ToPNoC VII, LNCS 7480, pp. 56-115

5. **Choppy C. et. al.** The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification // PETRI NETS 2010, LNCS 6128, pp. 145-164

6. **CPN Tools Homepage** // [Электронный ресурс]. Режим доступа: http://cpntools.org/, свободный, (дата обращения: 25.05.2014)

7. **Evangelista S.** High Level Petri Nets Analysis with Helena // ICATPN 2005, LNCS 3536, pp. 455-464

8. **Fronc L., Duret-Lutz A.** LTL Model Checking with Neco // ATVA 2013, LNCS 8172, pp. 451-454

9. **Fronc L., Pommerau F.** Towards a Certified Petri Net Model-Checker // APLAS 2011, LNCS 7078, pp. 322-336

10. **Holzmann, G. J.** The Spin model checker: primer and reference manual, Addison Wesley, 2003

11. **SPIN – Formal Verification** // [Электронный ресурс]. Режим доступа: http://spinroot.com/, свободный, (дата обращения: 25.05.2014)

12. **Карпов, Ю. Г.** Model Checking. Верификация параллельных и рапределённых программных систем, БХВ-Петербург, 2010

13. **Package javax.xml.parsers** // [Электронный ресурс]. Режим доступа: http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/package-summary.html, свободный, (дата обращения: 03.06.2014)

14. **Ахо А. В., Сети Р. Ульман Д. Д.** Компиляторы: принципы, технологии и инструменты, Вильямс, 2003

15. **Vizovitin N. V., Nepomniaschy V. A., Stenenko A. A.** Verification of UCM-

Specifications of Distributed Systems Using Coloured Petri Nets // Proc. of the Fourth Workshop “Program Semantics, Specification and Verification: Theory and Applications”, 2013, pp. 70-80

16. **Таненбаум Э.** Компьютерные сети. 5-е издание, Питер, 2012

17. **Cavalli A., Maag S.** A New Algorithm for Service Interaction Detection // Proc. ICFEM 2002. LNCS. 2002. V. 2495, pp. 371–382

18. **Keck D. O., Kuehn P. J.** The Feature and Service Interaction Problem in Telecommunications Systems: A Survey // IEEE Trans. on Software Eng. 1998. V. 24, No 10, pp. 779–796

19. **Стененко, А. А.** Верификация моделей распределённых систем, представленных раскрашенными сетями Петри. // Материалы 50ой Международной научной студенческой конференции «Студент и научно-технический прогресс»: Информационные технологии, НГУ, Новосибирск, 2012, с. 173.

20. **Стененко, А. А.** Верификация автоматных спецификаций с помощью раскрашенных сетей Петри. // Материалы 51ой Международной научной студенческой конференции «Студент и научно-технический прогресс»: Информационные технологии, НГУ, Новосибирск, 2013, с. 219.

# ПРИЛОЖЕНИЕ А. Грамматика языка описания автоматных спецификаций

Для описания грамматики используется расширенная нотация Бэкуса-Наура. Здесь заключённая в фигурные скобки последовательность нетерминалов и терминалов может быть повторена ноль или более раз, а заключённая в квадратные скобки последовательность может быть повторена ноль или один раз. Терминалы заключены в кавычки, а нетерминалы — в треугольные скобки. Терминалами являются также identifier и integer обозначающие буквенно-цифровой идентификатор и неотрицательное целое число в десятичной записи соответственно. Стартовым нетерминалом языка является model. Далее приведены все нетерминалы грамматики и содержащие их правила.

```
<model> ::= "{" <globals> <signal_types> <automate_types>
            <automate_instances> <initial_signals>
            <initial_statements> "}"
```

Нетерминал globals задаёт глобальные переменные.

```
<globals> ::= "{" { <variable_declaration> } "}"
```

Определению переменной соответствует нетерминал variable_declaration.

```
<variable_declaration> ::= <data_type_name> <variable_name>
                           { <array_dimension> }
                           [ <initializer> ]
```

Нетерминал data_type_name задаёт тип данных: целочисленный или булевский.

```
<data_type_name> ::= "int" | "bool"
```

Нетерминал array_dimension соответствует размерности массива. Индексы элементов массива принимают значения в интервале от нуля до числа не единицу меньшего, чем размер массива.

```
<array_dimension> ::= "[" <const_expression> "]"
```

Нетерминал initializer задаёт начальное значение переменной.

```
<initializer> ::= "=" <const_expression>
```

Нетерминал signal_types определяет имена типов сигналов, которые используется в

модели.

```
<signal_types> ::= "{" { <signal_type_name> } "}"
```

Нетерминал automate_types определяет классы автоматов.

```
<automate_types> ::= "{" { <automate_type> } "}"
```

Нетерминал automate_type соответствует определению автоматного класса.

```
<automate_type> ::= "{" <automate_type_name> <states> <locals>
                    <transitions> "}"
```

Нетерминал states определяет имена состояний автоматного класса.

```
<states> ::= "{" { <state_name> } "}"
```

Нетерминал locals определяет локальные переменные автоматного класса.

```
<locals> ::= "{" { <variable_declaration> } "}"
```

Нетерминал transitions соответствует переходам автоматного класса.

```
<transitions> ::= "{" { <transition> } "}"
```

Каждый из переходов задаётся нетерминалом transition. Для перехода определяется его имя, задаются его начальное и конечное состояния, множество выходных сигналов, охранное условие и вычисления, которые автомат данного класса выполнит при срабатывании данного перехода.

```
<transition> ::= "{" <transition_name> <state_name>
                 <state_name> <signals> <expression>
                 <statements> "}"
```

Нетерминал signals определяет множество выходных сигналов перехода.

```
<signals> ::= "{" { <signal> } "}"
```

Нетерминал statements соответствует вычислениям, изменяющим значения переменных.

```
<statements> ::= "{" { <statement> } "}"
```

Нетерминал automate_instances задаёт множество автоматов.

```
<automate_instances> ::= "{" { <automate_instance> } "}"
```

Нетерминал automate_instance определяет автомат. В определение автомата входят его

класс, выражение, задающее его идентификатор, начальное состояние, а также вычисления, устанавливающие начальные значения локальных переменных.

```
<automate_instance> ::= "{" <automate_type_name> <expression>
                        <state_name> <const_statements> "}"
```

Для задания переменным автомата начальных значений используется нетерминал const_statements.

```
<const_statements> ::= "{" { <const_statement> } "}"
```

Для задания множества сигналов, которые присутствуют в среде изначально, используется нетерминал initial_signals.

```
<initial_signals> ::= "{" { <const_signal> } "}"
```

Для задания начальных значений глобальным переменным используется нетерминал initial_statements.

```
<initial_statements> ::= "{" { <const_statement> } "}"
```

Нетерминал const_statement задаёт начальное значение одной скалярной переменной.

```
<const_statement> ::= <const_lvalue> "=" <const_expression>
```

В качестве скалярной переменной может использоваться элемент массива.

```
<const_lvalue> ::= <variable_name> { <dimension_const_expr> }

<dimension_const_expr> ::= "[" <const_expression> "]"
```

Нетерминал statement соответствует присваиванию скалярной переменной нового значения. Его отличием от const_statement является то, что при его вычислении могут использоваться значения переменных модели.

```
<statement> ::= <lvalue> "=" <expression>

<lvalue> ::= <variable_name> { <dimension_expr> }

<dimension_expr> ::= [ <expression> ]
```

Доступ к полям принимаемого сигнала в выражениях записывается как I.<имя поля>. Нетерминалы выражений определяются следующим образом:

```
<const_expression> ::= <constant> | "(" <const_operator>
                       { <const_expression> } ")"
```

```
<expression> ::= <const_expression> | <lvalue> |
                 "(" <operator> { <expression> } ")" |
                 "I.type" | "I.src" | "I.param" |
                 <signal_type_name>
<const_operator> ::= "+" | "-" | "~" | "*" | "/" | "%" |
                     "==" | "!=" | "<" | ">" | "<=" | ">=" |
                     "&&" | "||" | "!"
<operator> ::= <const_operator> | "random"
```

Сигналы задаются нетерминалами signal и const_ signal:

```
<const_signal> ::= "{" <const_expression> <const_expression>
                   <signal_type_name> <const_expression> "}"
<signal> ::= "{" <expression> <expression> <signal_type_name>
             <expression> "}"
 <signal_type_name> ::= <identifier>
```

В качестве имён переменных, классов автоматов, состояний и переходов используются идентификаторы, состоящие из латинских букв и цифр, а также символа подчёркивания. Идентификаторы не могут начинаться с цифры. Различающиеся регистром буквы считаются различными:

```
<variable_name> ::= identifier
<automate_type_name> ::= identifier
<state_name> ::= identifier
<transition_name> ::= identifier
```

В качестве констант могут использоваться неотрицательные целые числа в десятичной записи или булевские значения.

```
<constant> ::= integer | "true" | "false"
```