Российская академия наук
Сибирское отделение
Институт систем информатики
имени А. П. Ершова

М. Ю. Машуков, Т. Г. Чурина

МОДЕЛИРОВАНИЕ СПЕЦИФИКАЦИЙ ЯЗЫКА SDL С ПОМОЩЬЮ РАСКРАШЕННЫХ СЕТЕЙ ПЕТРИ

Препринт
144

Новосибирск 2007

В работе рассматриваются SDL-спецификации распределенных систем с динамическим порождением и уничтожением экземпляров процессов. Для них предложен метод трансляции в сети Йенсена, сетевая модель исследуется в системе CPN Tools. Описана реализация метода трансляции и приведены оценки размера сети.

Эта работа частично поддержана грантом РФФИ № 07-07-00173а.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Mikhail Mashukov, Tatyana Churina

MODELING SDL-SPECIFICATIONS VIA COLORED PETRI NETS

Preprint
144

Novosibirsk 2007

A translation method from SDL into coloured Petri nets and its implementation are presented. The net model is investigated in the CPN Tools system. Complexity bounds of the resulted nets that confirm effectiveness of the translation method are given.

This work is partly supported by RFBR grant N 07-07-00173a.

ВВЕДЕНИЕ

Анализ, валидация и верификация коммуникационных протоколов — актуальная проблема современного программирования.

Один из подходов к решению этой проблемы состоит в автоматическом переводе спецификаций протоколов в формальные модели [3, 5, 11, 14], для которых существуют эффективные методы анализа и автоматические средства верификации. Известны примеры трансляции спецификаций в конечно-автоматные модели [11], различные модели сетей Петри [1, 2, 7, 8, 9, 12], алгебры процессов [10] и темпоральные логики действий [13].

Стандартный язык выполнимых спецификаций SDL широко используется для представления коммуникационных протоколов. Опубликован ряд работ по трансляции SDL-спецификаций в различные модели сетей Петри. Среди сетей Петри высокого уровня можно выделить раскрашенные сети Петри (РСП) [14, 15], принятые в качестве международного стандарта, для которых разработаны и реализованы практические методы анализа. Кроме того, существуют системы Design/CPN [15] и CPN Tools [6], активно используемые в практических исследованиях.

В ИСИ СОРАН ведется работа по отображению языка SDL в раскрашенные сети Петри. Был разработан и реализован алгоритм [4, 20, 21] трансляции SDL-спецификаций с динамическими конструкциями в сети Йенсена. При этом полученная сетевая модель исследовалась в системе Design/CPN. Также было разработано отображение SDL-спецификаций без динамических конструкций в модифицированные раскрашенные сети — *иерархические временные типизированные (ИВТ-сети)* [16, 18, 19].

Настоящая работа является логическим продолжением [4] и предлагает способ перевода SDL-спецификаций с динамическим порождением и уничтожением экземпляров процессов в РСП. Сетевая модель исследуется в системе CPN Tools.

Данная работа состоит из четырех разделов. Первые два содержат описания базовых понятий языка SDL и сетевой модели. В третьем разделе изложены основные принципы отображения SDL-спецификаций в РСП, в четвертом — приводится описание алгоритма оптимизации и оценка размера сети.

1. ОБЗОР ЯЗЫКА SDL

SDL [17, 18] — язык спецификации и описаний — разработан бывшим Международным консультативным комитетом по телеграфии и телефонии (CCITT) и предназначен для описания структуры и функционирования систем реального времени, в частности сетей связи. SDL построен на базе модели конечного автомата по объектно-ориентированной схеме.

Самый общий объект, описываемый на SDL, называется *системой.* Все остальные объекты находятся на более низких уровнях иерархии определений. Все, что не вошло в описание системы, называется окружением (внешней средой) системы. Каждая система должна иметь уникальное имя.

Важной особенностью языка SDL является концепция типов данных, в основу которой положена алгебраическая модель. Все виды данных, используемые в конкретной системе, рассматриваются как компоненты единого типа. Элемент типа данных называется значением.

Все значения типа данных определяют множество, на котором задаются операторы. Константа (литерал в SDL) является 0-местным оператором; n-местный оператор (где $n \geq 1$) определяется своей сигнатурой, состоящей из имени оператора, типа результата и типов параметров. Действие оператора задается алгебраическими правилами — аксиомами. Совокупность множества значений типа данных, операторов со своими сигнатурами и аксиом образует алгебраическую систему.

В любой SDL-системе существуют предварительно определенные типы (сорта в SDL), такие как целые и вещественные числа, символы, строки. Другие сорта, например различные виды массивов, генерируются по шаблонам с помощью уже определенных сортов, операторов и аксиом.

В SDL определены две синтаксические формы описания систем. Одна — текстовая, совпадающая с формой описания обычных языков программирования, другая — графическая, в которой система описывается в виде диаграмм, состоящих из графических символов. Текстовая форма описания более богата, графическая — более наглядна.

Система состоит из одного или нескольких *блоков*, соединенных между собой и с окружающей средой *каналами*, по которым передаются *сигналы*. Из окружения система получает внешние сигналы и в окружение возвращает ответы, запуск системы возможен только по сигналу извне. Каждый блок и канал в системе имеют уникальное имя.

Каналы бывают одно- или двусторонними. При каждом канале должны быть указаны имена всех сигналов, которые этот канал может передавать. При сигнале могут быть указаны имена сортов, таким образом, сигнал мо-

жет нести с собой значения указанных сортов. Несколько сигналов могут быть объединены в один список, которому присваивается уникальное имя. Один сигнал может входить в разные списки. Сигналы в каналах могут задерживаться, создавая очередь по мере поступления сигналов.

Средствами SDL обеспечивается многоуровневое описание системы. По мере «спуска» от уровня к уровню либо детализируются описания уже имеющихся в системе объектов, либо вводятся новые объекты. Блок может быть разбит на более мелкие единицы — подблоки, которые, в свою очередь, сами являются блоками. Подблоки соединяются внутренними каналами между собой и с рамкой блока. Будем называть внешними каналы, входящие в блок или выходящие из блока. В разбиении блока для внешних каналов должны быть указаны имена внутренних каналов, которые подсоединены к внешним. Присоединение происходит таким образом, что каждый сигнал, поступивший по внешнему каналу, должен передаваться только по одному внутреннему каналу, внутренний канал не может передавать сигнал, который не передавался по внешнему каналу. Аналогичными средствами осуществляется разбиение канала на подканалы и новые блоки.

Рассмотрим спецификацию системы `S`, текстовое описание которой приведено ниже, графическое на рис. 1.

```
system S;
  newtype CommandType
    literals
      Init, PrioritySend, NormalSend, Close;
  endnewtype TPacketType;
  signal data(Integer), control(CommandType),
     conf(Integer), packet(PId, Integer);
  channel CInt
    from Users to Router with conf, control, data;
    from Router to Users with conf;
  endchannel CInt;
  channel COut1
    from Router to env with packet;
    from env to Router with packet;
  endchannel COut1;
  channel COut2
    from Router to env with packet;
  endchannel COut2;
  block Users referenced;
  block Router referenced;
endsystem S;
```

В ней описаны три канала — CInt, COut1 и COut2 и два блока — Users и Router. По двунаправленному каналу CInt, соединяющему блоки между собой, могут передаваться сигналы conf, control и data от блока Users к блоку Router и сигнал conf — в обратном направлении. Канал COut1 также является двунаправленным, он соединяет блок Router с окружением и может передавать сигнал packet. Канал COut2 — однонапраленный. Сигнал data несет значение целого сорта, control — значение определенного пользователем сорта CommandType, сигнал packet — идентификатор процесса и целое значение. Блоки Users и Router в системе не описаны.

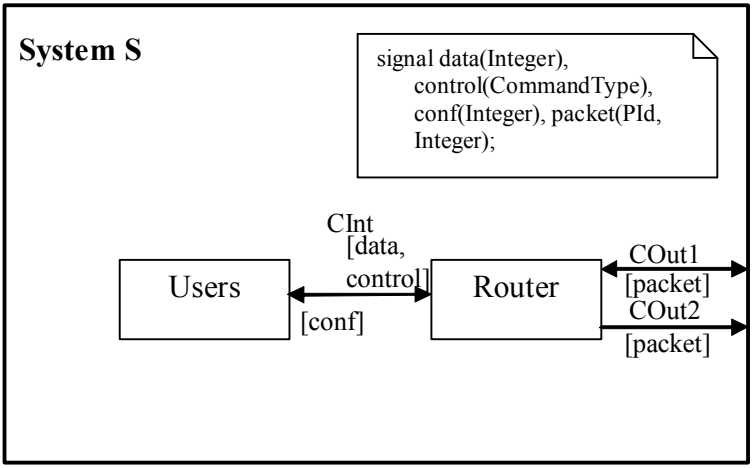


Рис. 1. Описание системы S

Система представляет собой модель пользователей, передающих данные в сеть. Блок Users содержит описание процесса пользователя. Блок Router моделирует некий прибор, который реализует передачу данных между пользователями и окружением посредством внешних каналов. В модели имеется два канала передачи данных — COut1, надежный и дорогой канал и COut2 — более дешевый и менее надежный канал, обеспечивающий передачу данных только в одном направлении.

На самом нижнем уровне иерархии определений блоки содержат процессы, являющиеся функциональными компонентами системы и определяющие ее поведение. Внутри блока маршруты связывают процессы между

собой и с рамкой блока. При графическом изображении маршруты, которые связывают процессы с рамкой блока, должны либо начинаться от той точки, в которой в блок входит канал, либо оканчиваться в той точке, в которой из блока выходит канал. К одному входному/выходному каналу могут быть присоединены начальные/конечные точки нескольких маршрутов. Распределение сигналов по маршрутам, присоединенным к некоторому каналу, происходит таким образом, что по каждому маршруту должен передаваться хотя бы один сигнал, поступающий по каналу, а каждый сигнал, поступающий в канал, должен передаваться хотя бы по одному присоединенному маршруту. Кроме того, маршрут не может передавать сигнал, который не передавался по каналу.

Описание процесса состоит из трех частей: заголовка, декларативной части и тела процесса. Заголовок процесса содержит имя, число экземпляров и список формальных параметров с указанием их типов. Формальные параметры — это переменные, используемые в теле процесса. Число экземпляров — пара целых чисел, первое из которых указывает количество создаваемых экземпляров процессов в момент инициализации системы, второе задает максимальное число экземпляров процессов, которые могут существовать в блоке одновременно. Так как по одному описанию процесса может быть создано несколько экземпляров, то каждый экземпляр должен получить персональный идентификатор (ПИД).

Для этой цели каждый экземпляр процесса наделяется переменной `SELF`, которой присваивается личный идентификатор созданного экземпляра. Кроме того, каждый экземпляр обладает переменной `SENDER`. Когда процесс-получатель воспринимает входной сигнал, этой переменной присваивается личный идентификатор экземпляра процесса-отправителя.

Декларативная часть процесса содержит описания констант, типов, переменных, экспортируемых переменных, переменных обозревания, входных и выходных сигналов, списков сигналов, таймеров, процедур и макрокоманд.

Тело процесса описывает действия, которые совершает процесс под влиянием входных сигналов. Процесс либо находится в одном из своих состояний, либо совершает переход. При этом каждое состояние должно иметь свое уникальное по отношению к этому процессу имя. Описание каждого перехода, не входящего в конструкцию «непрерывный сигнал», начинается с ключевого слова `INPUT`. Ключевые слова `JOIN`, `STOP`, `NEXTSTATE` (если последнее опущено, то `STATE`) называются «замыкателями» перехода.

Процесс имеет одну стартовую вершину, за которой следует переход, совершающийся не под влиянием входного сигнала, а в результате возникновения процесса. Дальнейшие переходы из состояния в состояние возможны только под воздействием входных сигналов, исключение составляют переходы, входящие в конструкцию «непрерывный сигнал».

Все сигналы, пришедшие в порт процесса, образуют в нем очередь. Процесс, находясь в одном из своих состояний, обращается к очереди сигналов. Если очередь пуста, процесс ждет, иначе, выполняет переход. Для каждого состояния процесса однозначно указывается, как должен реагировать процесс на любой сигнал, который стоит первым в очереди. Возможны три ситуации:

- явно указано, какой переход должен совершить процесс. Тогда процесс удаляет из очереди сигнал и начинает указанный переход;
- явно указано, что восприятие сигнала должно быть отложено до того, как процесс войдет в следующее состояние. Тогда процесс оставляет сигнал на своем месте в очереди и переходит к обработке следующего сигнала. Это действие называется сохранением сигнала — `SAVE`;
- нет никакого указания на то, как должен реагировать процесс на сигнал. В этом случае сигнал удаляется из очереди, и процесс переходит к обработке следующего сигнала.

Конструкция «непрерывный сигнал» содержит переход с входными сигналами, назовем его входным переходом, и переходы, каждый из которых начинается не с ключевого слова `INPUT`, а с булевского выражения с приоритетом. Находясь в некотором состоянии, процесс просматривает очередь сигналов. Если она пуста (т. е. в ней нет сигналов, указанных во входном переходе), то проверяются булевские выражения, входящие в эту конструкцию. Если истинно только одно из них, то выполняется последовательность действий, следующая за этим булевским выражением, иначе выполняется последовательность действий, следующая за выражением, имеющим максимальный приоритет. Если в очереди находятся сигналы, отличные от сигналов, указанных во входном переходе, то под их воздействием выполняется пустой переход и они удаляются из очереди.

При переходе процесс может выполнить такие действия, как присваивание, отправка сигнала, принятие решения, безусловный переход к другой последовательности действий, запрос на создание другого процесса, экспортно-импортную операцию, установку и сброс таймера.

SDL-спецификация может содержать операторы отправления сигналов следующих видов:

1) OUTPUT <имя сигнала> [<параметры>] TO <личный идентификатор>

2) OUTPUT <имя сигнала> [<параметры>] TO <имя процесса>

3) OUTPUT <имя сигнала> [<параметры>] TO this

4) OUTPUT <имя сигнала> [<параметры>]

5) OUTPUT <имя сигнала> [<параметры>] [TO <л. и. или имя пр.>] VIA <имя маршрута>, … [<имя канала>, … <имя маршрута>]

6) OUTPUT <имя сигнала>[<параметры>] VIA ALL (<имя маршрута>, … [<имя канала>, … <имя маршрута>])*

Приведенные выше конструкции соответственно означают, что сигнал, возможно с параметрами, посылается соответственно:

1) экземпляру процесса с личным идентификатором, указанным после служебного слова TO;
2) любому (одному) экземпляру процесса с указанным именем;
3) любому (одному) экземпляру процесса, выполняющего оператор OUTPUT;
4) одному из экземпляров процессов, связанных с процессом-отправителем маршрутами, которые могут передавать этот сигнал;
5) одному из экземпляров процессов, с которыми процесс-отправитель связан поименованными в конструкции (с помощью VIA) маршрутами;
6) по каждому пути (список вида <имя маршрута>, … [<имя канала>, … <имя маршрута>]), указанному после служебных слов VIA ALL. В этом случае посылается несколько экземпляров сигнала. Каждый экземпляр воспринимается только одним из экземпляров процесса, находящимся в конце указанного пути. Если некоторый процесс не имеет экземпляров в момент отправки сигнала, соответствующий экземпляр сигнала удаляется.

Заметим, что сигналы всегда несут ПИД экземпляра-отправителя.

Последовательность действий, выполняемых процессом во время перехода, может иметь разветвленную структуру. Для этого осуществляется проверка истинности некоторого выражения в конструкции принятия решения — DECISION, которая состоит из вопроса и не менее двух ответов.

После каждого ответа указывается последовательность действий, которую должен выполнить процесс. Если после нескольких ответов должна быть выполнена одна и та же последовательность действий, то эти ответы объединяются. Также возможен только один ответ ELSE, охватывающий все значения выражения (стоящего в вопросе), не учтенные во всех остальных ответах. Конструкция DECISION замыкается ключевым словом ENDDECISION. Если последовательность действий не заканчивается замыкателем перехода, то переход продолжается тем предложением, которое стоит непосредственно после ENDDECISION.

Безусловный переход к другой последовательности действий осуществляется конструкцией JOIN, аналогичной оператору GOTO в языке Паскаль. С ее помощью определяется, что следующим должна выполняться последовательность действий этого же самого процесса, перед которым стоит указанная в конструкции JOIN метка.

В языке SDL процессы порождаются либо в момент инициализации системы, либо один процесс порождает другой во время функционирования системы. Порождение одного процесса другим осуществляется оператором

CREATE <имя_процесса> <фактические_параметры>,

который называется запросом на порождение, где <имя_процесса> — имя того описания процесса, по типу которого создается экземпляр процесса. Фактических параметров должно быть столько же, сколько формальных в порождаемом процессе, и их сорта должны совпадать с сортами соответствующих формальных параметров. Каждый экземпляр процесса обладает переменными PARENT и OFFSPRING. Переменной PARENT порожденного процесса присваивается значение переменной SELF родительского процесса. Переменной OFFSPRING родительского процесса присваивается значение переменной SELF его последнего отпрыска.

В каждом блоке должен быть хотя бы один процесс, возникающий при инициализации системы. Этот процесс может порождать другие процессы, но только в своем блоке. Для порождения процесса в другом блоке необходимо отправить соответствующий сигнал «процессу-производителю» этого блока. С момента возникновения экземпляры процесса начинают функционировать независимо друг от друга. Сигналы, посланные одним экземпляром процесса другому экземпляру этого же процесса («брату»), помещаются непосредственно в порт «брата», так как между ними нет маршрутов.

Один экземпляр процесса может передавать различным экземплярам других процессов значение некоторой переменной с помощью экспортно-импортной операции. Для этого при описании переменной он должен ука-

зать, что ее значение в дальнейшем будет экспортироваться. Если другой процесс хочет получить экспортированное значение переменной, то он должен в своей декларативной части указать, что намеревается импортировать это значение.

Назначение языка SDL — описание систем реального времени. Поэтому комплекс, реализующий систему, должен обладать средствами управления временем, например, часами, которые показывают абсолютное время в согласованных единицах времени. В языке предусмотрена стандартная функция `NOW`, значением которой является значение текущего момента абсолютного времени, и имеется возможность описания таймеров и установки в них любого времени, указанного в принятых единицах. Реализуется эта возможность оператором `SET`. Когда абсолютное время в системе станет равным установленному в таймере, в порт процесса будет установлен сигнал от таймера, имя которого совпадает с именем самого таймера. Однако такой сигнал устанавливается только в том случае, если за это время таймер не подвергался переустановке. Чтобы процесс мог воспринять от таймера сигнал, последний должен быть указан в теле процесса в качестве входного. Таймер считается активным с момента установки и до момента восприятия от него сигнала. Перевод таймера в неактивное состояние («сброс таймера») осуществляется оператором `RESET`.

Процессы могут содержать описания и вызовы процедур. Один процесс может вызвать процедуру, описанную в другом процессе, но в этом же блоке. При описании процедур явно указывается способ передачи параметров посредством ключевых слов `IN` и `IN/OUT`. После слова `IN` указываются параметры, которые будут переданы процедуре по значению, после слов `IN/OUT` — по ссылке. Рекурсивные процедуры в языке SDL не допускаются. Процедуры также не могут содержать экспортируемых и обозреваемых переменных. Одну и ту же процедуру могут одновременно вызывать несколько процессов. Вызов процедуры выполняется следующим образом. Создается экземпляр-копия вызванной процедуры. В нем каждому формальному параметру, передающемуся по значению, присваивается значение соответствующего фактического параметра. Формальные параметры, передающиеся по ссылке, заменяются соответствующими фактическими параметрами. После выполнения процедуры созданный экземпляр прекращает свое существование, результат его работы передается вызывающему процессу в виде значений, присвоенных фактическим переменным, переданным по ссылке.

Язык SDL позволяет использовать макросредства. Механизм макросредств представлен конструкциями определения и вызова макрокоманды.

Определение макрокоманды состоит из заголовка и тела макрокоманды. Заголовок имеет имя и список формальных параметров, которые при подстановке заменяются текстом действительных параметров. Тело макрокоманды задается последовательностью предложений языка SDL. Макрокоманда — это сокращенное обозначение отрезка текста, который не может функционировать самостоятельно, а должен быть вставлен в тело вызывающего процесса. При вызове создается копия тела макрокоманды, в которой все формальные параметры заменяются соответствующими лексическими единицами, указанными в вызове макрокоманды. Из описания системы удаляется вызов макрокоманды, на его место вставляется указанная копия тела макрокоманды. Определения макрокоманд помещаются в декларативной части описания системы, блока или процесса. Определение макрокоманды не может быть вложенным в определение другой макрокоманды.

2. СЕТЕВАЯ МОДЕЛЬ

Ординарную сеть Петри можно определить как размеченный ориентированный граф с вершинами двух типов: *местами* и *переходами*, соединенными дугами таким образом, что каждая дуга соединяет вершины различных типов [14, 15]. Для изображения перехода используется, как правило, прямоугольник или вертикальная черта, места — окружность, а дуги — направленная стрелка. Места помечаются целыми неотрицательными числами — *разметка места*. В графическом представлении сети разметка изображается соответствующим числом точек (фишек) в месте.

Вершина x называется *входной* для вершины y, если в сети существует дуга, ведущая от вершины x к вершине y. Аналогично, вершина x называется *выходной* для вершины y, если в сети существует дуга, ведущая от вершины y к вершине x. Дуги, ведущие к вершине x и от нее, называются соответственно *входными* и *выходными* дугами этой вершины.

Сеть Петри функционирует, переходя от разметки к разметке. Функционирование начинается при заданной начальной разметке. Смена разметок происходит в результате срабатывания одного из переходов. Переход может сработать при некоторой разметке, если все входные места перехода содержат хотя бы по одной фишке. Срабатывание перехода изымает по фишке из каждого входного места перехода и помещает по фишке в каждое его выходное место.

Таким образом, сеть Петри моделирует некоторую систему и динамику ее функционирования. При этом места и находящиеся в них фишки представляют состояние моделируемой системы, а переходы — изменение ее состояний.

Неиерархические раскрашенные сети являются расширением базовой модели сетей Петри [14]. В отличие от ординарных сетей Петри каждая фишка в раскрашенной сети обладает индивидуальностью — значением некоторого типа, которое называется *цветом*. Неиерархическая раскрашенная сеть состоит из трех частей: структуры сети, деклараций и пометки сети.

Иерархическая раскрашенная сеть представляет собой множество неиерархических сетей, связанных между собой по определенным правилам и функционирующих как единое целое.

Декларации состоят из описания множеств цветов (типов) и объявления переменных, каждая из которых принимает значения из некоторого множества цветов. Декларации также могут содержать определение операций и функций. Располагаются декларации, как правило, в верхней части страницы в прямоугольнике, границы которого изображаются пунктирной линией. В иерархических раскрашенных сетях декларации общих для всех страниц множеств цветов и переменных часто выносят на отдельную страницу. Декларации сети будем изображать курсивным шрифтом.

В раскрашенных сетях определены четыре базовых множества цветов, соответствующие стандартным типам: целый, вещественный, строковый и булевский. Базовым также является специальное множество цветов, состоящее из одного элемента, которое описывается как *color E = with e*. Фишки со значением *e* будем называть *бесцветными*.

Множество цветов может описываться путем перечисления всех возможных значений. Также в раскрашенных сетях имеется несколько механизмов, обеспечивающих возможность конструирования нового множества цветов из уже продекларированных множеств. При моделировании SDL-спецификаций, в основном, будем использовать — *product* и *list*.

Декларация *color P = product A * B * C* определяет множество цветов, состоящее из всех троек вида *(a, b, c)*, где *a* — значение из *A*, *b* — из *B* и *c* — из *C*. Такие множества цветов будут использоваться для представления записей в спецификациях.

Согласно декларации *color L = list A,* элементами множества цветов *L* являются все списки, состоящие из элементов множества *A*. Значения последнего множества цветов представляются как *[v_1, v_2,...,v_n]* либо в виде выражения *h::t*, где *h* — первый элемент, или *голова*, списка, а *t* —

список без первого элемента, или *хвост* списка. Для пустого списка используются обозначения *[]* или *nil*.

В раскрашенных сетях для базовых множеств цветов существуют стандартные операции и функции, применимые к элементам множества, которые не требуется явно описывать в декларациях. Например, над списками определены операции: конкатенация двух списков (l_1^^l_2), взятие головы списка (*hd l*), хвоста списка (*tl l*) и определение длины списка (*length l*).

Пометка сети приписывается месту, переходу либо дуге и описывает правила распределения и перемещения фишек по сети. Каждое место имеет три разных типа пометок: имя места, множество цветов и инициализирующее выражение. Имя не имеет формального значения и служит для идентификации. Множество цветов определяет тип фишек, которые могут находиться в месте, т. е. любая фишка, находящаяся в месте, должна иметь цвет, который является элементом данного множества цветов. Инициализирующее выражение определяет начальную разметку места.

Переходы имеют два типа пометок: имена и спусковые функции, а дуги имеют один тип — выражения. Спусковая функция перехода является логическим выражением, которое должно быть выполнено до того, как переход сможет сработать.

Выражения на дугах могут содержать переменные, константы, функции и операции, определенные в декларациях. Все переменные, входящие в спусковую функцию перехода и выражения на связанных с ним дугах, будем называть *переменными перехода*.

Пример раскрашенной сети приведен на рис. 2. Сеть содержит три перехода *t1, t2* и *t3* и место *p,* которое является как входным, так и выходным местом переходов *t1* и *t3*, и — выходным для *t2*. Декларации сети содержат определение множества цветов *pair* и декларации переменных, входящих в выражения на дугах. Рядом с местом указаны имя места, связанное с ним множество цветов и начальная разметка места. В примере множество цветов есть *pair,* начальная разметка — *2`(1,0)@1,* что означает, что в месте *p* находятся две фишки — кортежи, в первой позиции которых содержится 1, а во второй — 0. Выражение, которое следует после символа @, относится к временному механизму и будет рассмотрено ниже, также как и выражение на дуге из перехода *t3*.

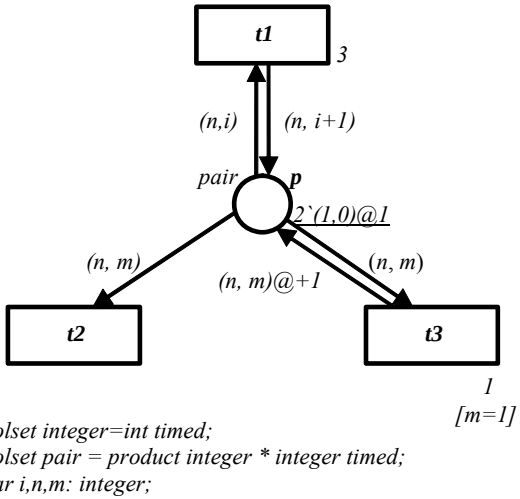


Рис. 2. Пример раскрашенной сети

Внутри переходов указаны их имена, справа от переходов — временные задержки и спусковая функция. Например, у перехода *t3* временная задержка равна 1, а спусковая функция — это выражение *[m = 1]*.

Далее в работе декларации могут опускаться. Множества цветов, приписанные местам, описываются в тексте; для переменных, входящих в выражение на дуге, предполагается, что они принимают значения из множества цветов, приписанного месту, с которым связана дуга. Например, дуга, ведущая из места *p* в переход *t2*, помечена выражением *(n,m)*. При срабатывании перехода *t2* из места *p* изымается фишка, переменной *n* присваивается значение первой позиции кортежа, а переменной *m* — значение второй позиции этого кортежа. На всех входных дугах одного перехода в первой позиции кортежа будем использовать одну и ту же переменную.

Функционирование раскрашенной сети отличается от функционирования ординарной сети тем, что возможность срабатывания перехода зависит как от наличия фишек во входных местах перехода, так и от их значений. Чтобы говорить о срабатывании перехода, необходимо определить значения переменных перехода. При этом все вхождения одной и той же переменной замещаются одним и тем же значением. Набор значений переменных, при которых выполнена спусковая функция перехода, называется *связыванием*. Значение выражения на дуге при выбранных значениях перемен-

ных определяет фишку (мультимножество фишек в общем случае), которая может быть «перемещена» по этой дуге. Другими словами, значение выражения на входной дуге определяет, сколько и каких фишек должно содержаться в соответствующем входном месте перехода, чтобы переход мог сработать при выбранных значениях переменных. Выражения на выходных дугах определяют, сколько и каких фишек будет помещено в выходные места перехода, когда он сработает.

Переход раскрашенной сети *возможен*, если можно выбрать такие значения переменных перехода, что в каждом входном месте перехода имеется фишка, определенная значением выражения на соответствующей дуге. Возможный переход может сработать. Срабатывание перехода изымает фишки из его входных мест и добавляет в выходные места. Количество и цвет изымаемых/добавляемых фишек определяются выражениями на соответствующих дугах.

Иерархическая раскрашенная сеть — это композиция множества неиерархических сетей, называемых *страницами*. Страницы могут содержать вершины специального типа, которые называются *модулями* и соединяются с местами на странице по тому же принципу, что и переходы.

Модуль представляет подсеть, располагающуюся на отдельной странице, которая, в свою очередь, может содержать модули. Такая страница называется *подстраницей* страницы, на которой располагается модуль. Подстраница содержит копии всех мест, с которыми связан модуль. Место-копия может быть входным для некоторого перехода или модуля на подстранице тогда и только тогда, когда его *прототип* является входным местом для модуля, представляющего подстраницу. Аналогично, только копия выходного места-прототипа может быть выходным местом некоторого перехода или модуля на подстранице. Место-прототип и его копия являются образами одного и того же «концептуального» места. Они имеют всегда одинаковую разметку.

Поведение иерархической сети эквивалентно поведению неиерархической сети, получающейся при замещении всех модулей страницами, которые они представляют. При этом каждый модуль вместе со своими дугами удаляется со страницы, а на его место помещается подсеть, располагавшаяся на подстранице. Соединение сетей происходит по местам: каждое место-прототип склеивается со всеми своими копиями.

Для моделирования таймерных конструкций в раскрашенных сетях введен временной механизм [14, 15]. Для представления времени введено понятие *глобальных часов*. Значение часов представляет текущее время в модели или *модельное время*. Начальное модельное время, т. е. значение ча-

сов, при котором сеть начинает функционировать, задается при описании сети. Множества цветов (все или часть) получают признак *timed*.

Фишка, принимающая значения из множества, обладающего признаком *timed*, дополнительно несет временное значение, называемое *временным штампом*. Оно определяет момент времени, раньше которого фишка не может использоваться при срабатывании какого-либо перехода. Разметка места *p* (рис. 2) вида 2`(1,0)@1 соответствует наличию в этом месте двух фишек цвета *pair* со значением (1,0), несущих временной штамп, равный 1. Фишка, принимающая значения из множеств, не имеющих признака *timed*, временного штампа не несет; это означает, что она готова к использованию с момента своего создания.

Временной штамп новой фишки вычисляется как текущее время в модели плюс задержка, величина которой определяется выражением @+*T*. Это выражение называется *временной пометкой*. Если временная пометка связана только с переходом, то во все выходные места перехода помещаются фишки с одинаковыми временными штампами. Временная пометка на выходной дуге определяет значение временного штампа только для фишки, помещаемой в место, для которого данная дуга входная. Если временная пометка присутствует и на переходе, и на дуге, то значения задержек суммируются.

На рис. 2 присутствуют оба типа пометок. При срабатывании переход *t1* помещает в место *p* фишку с временным штампом, равным значению глобальных часов, увеличенному на три единицы. Дуга от перехода *t3* к месту *p* имеет собственную временную пометку, которая увеличивает значение временного штампа, определенного переходом, на единицу. В результате, для фишки, помещаемой в место *p*, временной штамп равен текущему модельному времени плюс две единицы.

Будем называть раскрашенную сеть, которая получается из временной удалением всех временных пометок, *базовой* для временной сети. Во временных раскрашенных сетях переход называется возможным по цвету при выбранном связывании, если он возможен при этом связывании в базовой сети и временные штампы фишек, необходимых для выбранного связывания, должны быть не больше, чем текущее модельное время. Переходы, для которых выполнено последнее условие, называется *готовыми.* Срабатывание перехода происходит мгновенно.

Текущее модельное время не изменяется до тех пор, пока остаются готовые переходы. Когда в сети остаются только возможные по цвету переходы, происходит изменение значения глобальных часов. При этом новым

значением часов будет ближайший момент времени, в который какой-либо из возможных переходов становится готовым.

После инициализации сети на рис. 2 модельное время равно 0. В этот момент обе фишки в месте *p* недоступны, так как их временные штампы равны 1. Поэтому никакой переход в сети не может сработать. В сети произойдет изменение модельного времени, оно станет равно 1. При таком значении глобальных часов переходы *t1* и *t2* становятся готовыми. При срабатывании перехода *t1* может быть использована любая из фишек в месте *p*. В результате срабатывания перехода *t1* из места *p* изымается одна (любая) фишка и помещается новая фишка со значением *(1,1)* и временным штампом, равным 4. Срабатывание перехода *t2* приводит к тому, что из места удалится одна (любая) фишка с временным штампом, равным 1.

Для обеспечения доступа к фишкам, временной штамп которых больше, чем текущее модельное время, на входных дугах перехода используется временная пометка *@+[t]*. Такая пометка означает, что фишка доступна для этого перехода на *t* единиц времени раньше, чем указано во временном штампе фишки.

3. ПЕРЕВОД SDL-СПЕЦИФИКАЦИЙ В РСП

Сеть, моделирующая SDL-систему, строится с помощью поэтапного уточнения. На первом этапе создается страница, которая соответствует основной структуре системы и содержит по одному модулю для каждого описания блока. На втором этапе для каждого из полученных модулей строится дерево страниц, повторяющее иерархию блока с корнем в соответствующем блоке. На следующих этапах создаются сети, соответствующие процессам, затем — SDL-переходам и процедурам.

Маршруты и каналы спецификации в большинстве случаев не отображаются в места и переходы сети. Во внутреннем представлении транслятора создается граф связей блоков и процессов. По нему для каждого действия отправки сигнала определяются возможные процессы-получатели. Затем в сети строится конструкция, моделирующая добавление сигнала в нужный порт процесса.

Для представления стандартных типов — целый, булевский и вещественный — используются соответствующие множества цветов: *integer, Boolean* и *real*. Система CPN Tools не допускает прямого использования встроенных типов *int* и *bool* для определения сложных типов, поэтому в декларации добавляются следующие объявления:

colset integer=int timed;

colset Boolean=bool;

Заметим, что текущие версии CPN Tools не поддерживают работу с вещественным типом.

Перечислимые множества цветов сопоставляются перечислимым типам. Идентификаторы, используемые в качестве значений перечислимых типов в раскрашенных сетях, не могут повторяться. С этой целью при генерации сетевой модели создаётся перечислимый тип *allenums*, содержащий значения всех перечислимых типов в модели. Все перечислимые типы определяются как ограничение *allenums*. В любой модели *allenums* включает служебное имя *e*, используемое для обозначения состояния неинициализированного процесса, и *enum0*, используемое в качестве нулевого значения для перечислимых типов, определённых в SDL-спецификации.

В описании идентификаторов процессов SDL используется целый тип:

colset PIdType=int;

colset Receiver_PId=int;

Для определения структурированных типов данных мы допускаем использование массивов и записей.

Тип массив отображается в множество цветов типа список, к которому применяются две функции — чтение и запись элемента массива. Значение массива всегда представляется списком, содержащим N элементов, где $N = N_{max} - N_{min} + 1$, а N_{min} и N_{max} — соответственно минимальное и максимальное значения базового типа.

Записи в моделирующей сети представляются множествами цветов, полученных с помощью декларативной приставки *product*. Полям записи при таком представлении соответствуют элементы кортежа. Так, описания типов

```
newtype CommandType
  literals
    Init, PrioritySend, NormalSend, Close;
endnewtype;
newtype CommandData
  struct
    Sender Integer;
    Data Integer;
    Command CommandType;
endnewtype;
```

трансформируются в следующие множества цветов:

colset TPacketType = subset allenums with [enum0, Init, PrioritySend,
NormalSend, Close];
*color CommandData = product Integer * Integer * CommandType.*

Для доступа к полям записи требуется указывать имя переменной соответствующего типа в нужной позиции кортежа, например *(Sender, SenderData, CommType*), где *Sender* и *SenderData* — переменные типа *integer*, *CommType* — типа *CommandType*. Присваивание значения какому-либо полю записи в сети представляется кортежем на выходной дуге перехода, где в соответствующей этому полю позиции находится указанное значение. Например, операторам

```
TASK Comm!Sender := self;
TASK Comm!Data := 100;
Task Comm!Command := PrioritySend;
```

будет соответствовать кортеж *(self, 100, PrioritySend)*.

При отображении SDL-спецификации считаем, что все декларации вынесены на отдельную страницу. Определения новых типов преобразуются в множества цветов по описанной выше схеме. В процессе построения сети декларации дополняются переменными, которые входят в выражения на дугах и спусковые функции переходов.

3.1. Граф связей объектов SDL-спецификации

Несмотря на обилие возможностей маршрутизации сигналов в SDL — использование соединений каналов и маршрутов, конструкция `VIA` и т.д., для любого оператора отправки сигнала можно определить подмножество процессов, которое может получить отправляемый сигнал. Это подмножество не может измениться в процессе работы модели спецификации, и поэтому определяется статически, на этапе трансляции спецификации.

В процессе функционирования сетевой модели количество экземпляров процессов и их идентификаторы могут меняться, при этом конкретный экземпляр-получатель конкретного сигнала должен определяться уже на этапе симуляции модели.

Как уже было упомянуто выше, на этапе синтаксического анализа во внутреннем представлении транслятора создается граф связей объектов SDL-спецификации. Он представляет собой маркированный направленный граф, вершинами которого являются каналы, маршруты, блоки и процессы. Вершины могут соединяться одной или несколькими дугами, помеченными

типами передаваемых сигналов или символом *. Символ * используется для пометки дуг, по которым может передаваться любой сигнал.

В этом графе каждому блоку SDL-спецификации соответствует одна вершина. Она соединяется входными дугами с вершинами, соответствующими входным каналам и тем маршрутам блока, которые передают сигналы в окружение блока. Каждая выходная дуга вершины соответствует либо выходному каналу, либо маршруту, который передает сигналы от точки присоединения этого маршрута к каналу внутрь блока.

Каждому однонаправленному маршруту/каналу соответствует одна вершина. Двунаправленному каналу соответствует две вершины, каждая из которых представляет поток сообщений в одном направлении. В соответствии с терминологией SDL назовем их *входной* и *выходной* вершинами, соответственно. Входная вершина соответствует потоку сообщений к блоку, выходная — от блока.

Выходные дуги вершин, соответствующих маршрутам/каналам, помечаются сигналами, которые передаются по этим маршрутам/каналам в соответствующем направлении.

Каждому процессу соответствует одна вершина. Такие вершины содержат дополнительную информацию о сигналах, воспринимаемых процессом. Выходные дуги ведут к другим вершинам, соответствующим маршрутам, и содержат пометку *.

3.2. Моделирование структуры системы

При порождении сети, соответствующей структуре SDL-системы, используются видимые всем блокам описания сигналов и списков сигналов, описания каналов, соединяющих между собой блоки и окружающую среду и блоки, а также часть информации из описания процессов.

Строящаяся сеть содержит по одному модулю на каждый блок SDL-системы. Для большей наглядности в качестве имени модуля можно использовать имя соответствующего блока.

Для генерации персональных идентификаторов экземпляров процессов используется специальное место *NewPId*. Это место содержит одну фишку из множества цветов *integer*. Начальная разметка этого места — фишка со значением $n + 1$, где n — количество экземпляров процессов, которые создаются при инициализации системы. Это место становится входным и выходным для каждого модуля, представляющего блок, который содержит процессы, выполняющие запрос на создание другого процесса.

Для определения процесса-получателя по идентификатору экземпляра-получателя на этом этапе создается место *InstPIds,* которое содержит по одной фишке для каждого процесса. Изначально каждая фишка несет значение с идентификатором процесса и списком идентификаторов экземпляров этого процесса, создаваемых в момент инициации системы.

В результате построения графа связей и трансляции оператора OUTPUT (что будет описано ниже) на верхнем уровне будут созданы места, которые моделируют очереди сигналов, поступающих к процессам, находящимся внутри блоков системы.

Также на этом этапе для каждого однонаправленного канала, передающего сигналы из окружения/в окружение системы, создается одно место. Для двунаправленного канала создается два места, одно из них моделирует входной поток сигналов из окружения к блоку, другое — от блока к окружению.

Для представления очередей сигналов в каналах используются списки, таким образом, каждой очереди соответствует одна фишка. Поэтому каждое моделирующее канал место будет входным и выходным для модуля, соответствующего соединенному с этим каналом блоку. Начальная разметка места, представляющего канал, — фишка со значением *nil*.

Экспортно-импортная операция в языке SDL может производиться между процессами, принадлежащими различным блокам. Каждой экспортируемой переменной ставится в соответствие одно место, которое будет входным для модуля, отображающего блок с процессом «экспортер», и выходным для модуля, отображающего блок с процессом «импортер». Копии места, соответствующего экспортируемой переменной, появляются на страницах, связанных с модулями, представляющими процессы «экспортер» и «импортер». Инициализирующие выражения для мест, представляющих экспортируемые переменные, определяются из декларативной части процессов.

Пример

Рассмотрим спецификацию системы `S` (рис.1). Декларации сети при моделировании системы `S` дополняются следующими множествами цветов:

```
colset integer=int timed;
colset Boolean=bool;
colset RealType=integer;
colset PIdType=int;
```

colset Receiver_PId=int;
colset allenums = with e | enum0 | Init | PrioritySend | NormalSend | Close;
colset CommandType subset allenums with [enum0, Init, PrioritySend, NormalSend, Close];
*colset CInt_unit=record to:Receiver_PId * send:integer * sigm:allsigs * integer1:integer * CommandType1:CommandType * pid1:PIdType;*
colset CInt_type=list CInt_unit timed.
*colset COut1_unit=record to:Receiver_PId * send:integer * sigm:allsigs * pid1:PIdType * integer1:integer;*
colset COut1_type=list COut1_unit timed.
*colset COut2_unit=record to:Receiver_PId * send:integer * sigm:allsigs * pid1:PIdType * integer1:integer;*
colset COut2_type=list COut2_unit timed.

Также в декларации добавляются служебные переменные, используемые для связывания значений фишек, изымаемых/помещаемых из/в мест-каналов:

var CInt_l, CInt_l2: CInt_type;
var COut1_l, COut1_l2: COut1_type;
var COut2_l, COut2_l2: COut2_type;

В процессе дальнейшего построения сети декларации дополняются переменными, которые входят в выражения на дугах сети и в спусковые функции переходов, и, возможно, новыми множествами цветов, если блоки и процессы содержат собственные определения сортов, сигналов и списков сигналов.

Сеть для системы `S` приведена на рис. 3 (чтобы не загромождать рисунок, декларации сети опущены). Для большей наглядности в качестве имени модуля используется имя соответствующего блока.

Сеть содержит два модуля *User* и *Router* и 7 мест. Места *COut1_1* и *COut1_2* моделируют двунаправленный канал `COut1`, соединяющий блок *Router* с окружением. Место *COut2* моделирует однонаправленный канал `COut2`. Места *MQueue* и *UQueue* создаются в результате трансляции операторов `OUTPUT`, что будет описано ниже. Место *MQueue* моделируют очередь сигналов, поступающих к процессу, находящемуся внутри блока `Router`. Место *UQueue* моделируют очередь сигналов, поступающих к процессу, находящемуся внутри блока `Users`.

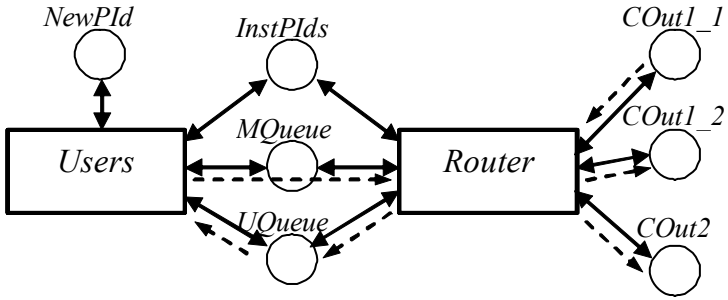


Рис. 3. Сеть для системы S

Как уже было указано выше, места, содержащие списки сигналов (*Mqueue, Uqueue, COut1_1, COut1_2, COut2*), соединяются с модулями двунаправленными дугами. Направления передачи сигналов показаны на рис. 3 пунктирными линиями, эти линии не являются элементами сети.

Таким образом, после завершения первого шага моделирования имеем сеть, которая состоит из модулей (представляющихся «черными ящиками»), соответствующих блокам в системе, и мест, полученных при отображении каналов. В данном примере нет экспортируемых переменных и временных конструкций.

3.3. Моделирование блока

Описание системы — это последовательность диаграмм, где каждая следующая диаграмма осуществляет дальнейшую детализацию системы. Предусмотрены диаграммы подструктур блоков и каналов. Диаграмма подструктуры блока используется в тех случаях, когда рассматриваемый блок представляет сложный объект и состоит из подблоков и внутренних каналов. Каналы внутри системы, которые соединяют блоки между собой и с окружением, называются *внешними*. В результате разбиения блоков возникает структура, аналогичная структуре системы, в которой рамка блока играет роль рамки системы.

На каждой странице, связанной с модулем, соответствующим некоторому блоку, отображается внутренняя структура этого блока. Это отображение осуществляется таким же способом, что и отображение структуры системы на первой странице. Каждому подблоку на подстранице

темы на первой странице. Каждому подблоку на подстранице соответствует один модуль.

Если описание какого-либо блока дано за пределами описания системы (указана ссылка на удаленное описание), то дальнейшего уточнения сети для этого блока не происходит. В результирующей сети данному блоку будет соответствовать один модуль, а именно тот, который порожден на предыдущем этапе построения. При моделировании подструктуры канала на странице, связанной с дополнительным модулем, отображается подструктура этого канала аналогично тому, как отображалась подструктура блока.

Каждый блок, не расчлененный на подблоки, должен содержать хотя бы один процесс. Процессы между собой и с рамкой блока соединяются маршрутами. Моделирование блока, состоящего из процессов, аналогично моделированию блока любого уровня иерархии, при этом каждому описанию процесса сопоставляется один модуль. Отличие состоит в моделировании точки присоединения маршрутов к каналу. Сигнал, поступающий к блоку по входному каналу, может передаваться по нескольким подсоединенным к нему маршрутам.

В описании семантики SDL точно не определяется, в какие маршруты будет передаваться сигнал. Каждый раз маршрут, по которому будет передаваться этот сигнал, и экземпляр процесса, который получит сигнал, выбираются произвольным образом из множества допустимых маршрутов и допустимых экземпляров. Маршрут является допустимым, если он может передавать сигнал, указанный в операторе отправки сигнала, и существует путь, по которому может быть передан этот сигнал от процесса-отправителя до точки входа в этот маршрут. Допустимыми экземплярами процесса являются все существующие в момент выполнения оператора отправки экземпляры всех процессов, являющихся выходными для допустимых маршрутов.

Соединение маршрутов и каналов в исходной спецификации отображается в графе связей объектов спецификации следующим образом. Для каждого входного маршрута, соединенного с однонаправленным каналом, в этом графе создается дуга, идущая из вершины, соответствующей данному каналу, в вершину, соответствующую данному маршруту. Маршруты `m1`, `m2` и `m3`, изображенные на рис. 4, являются, по терминологии SDL, входными к соответствующим процессам.

Для каждого выходного маршрута, соединенного с однонаправленным каналом, в графе создается дуга, идущая из вершины, соответствующей данному маршруту, в вершину, соответствующую каналу.

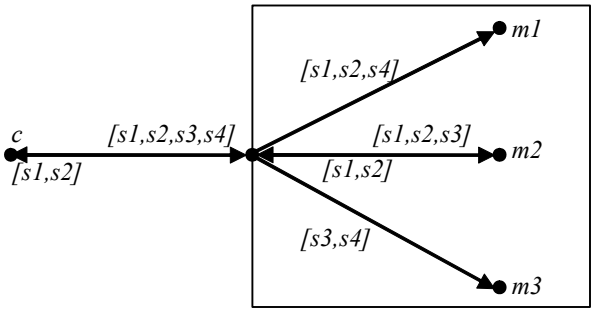


Рис. 4. Присоединение маршрутов `m1`, `m2`, `m3` к каналу `c`

При соединении маршрутов с двунаправленными каналами дуги в графе строятся следующим образом. А именно, при присоединении входного маршрута к каналу строится дуга, идущая из вершины, представляющей маршрут, во входную вершину, представляющую канал. При присоединении выходного маршрута к каналу строится дуга, идущая из вершины, представляющей маршрут, в выходную вершину, представляющую канал.

В случае соединения двунаправленного канала и двунаправленного маршрута создаются две дуги. Первая дуга идет из входной вершины, представляющей канал, во входную вершину, представляющую маршрут. Другая дуга идет из выходной вершины, представляющей маршрут, в выходную вершину, представляющую канал.

На рис. 4 показано присоединение маршрутов `m1`, `m2` и `m3` к каналу `c`. Канал может передавать в блок сигналы `s1`, `s2`, `s3` и `s4` и в обратном направлении — сигналы `s1` и `s2`. В маршрут `m1` из канала поступают сигналы `s1`, `s2` и `s4`, в `m2` — `s1`, `s2` и `s3`, а в маршрут `m3` — `s3` и `s4`. В обратном направлении — из маршрута `m2` в канал, передаются сигналы `s1` и `s2`. Присоединение маршрутов к процессам на рисунке не показано.

На рис. 5 представлен фрагмент графа связей, соответствующий описанной точке присоединения. Каналу `c` соответствуют вершины графа *c1* и *c2*, первая из которых моделирует направление передачи сигналов из окружения внутрь блока, а вторая — из блока в окружение. Однонаправленным маршрутам `m1` и `m3` соответствуют вершины графа *m1* и *m3*. Маршрут `m2` является двунаправленным и ему соответствует две вершины. Вершина *m2in* является входной и соответствует направлению передачи сигналов из окружения внутрь блока. Вершина *m2out* является

окружения внутрь блока. Вершина *m2out* является выходной и соответствует передаче сигналов в обратном направлении.

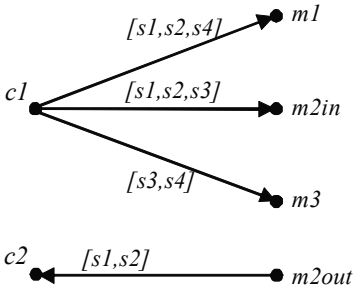


Рис. 5. Фрагмент графа, соответствующий точке присоединения

Пример

Рассмотрим описание блока `Router`.

```
block Router;
  signal prior_data(PId, Integer),
    norm_data(PId, Integer),
    switch_norm;
  signalroute Rin1
    from env to Marker with data, control;
  signalroute RInt2
   from env to Receiver with packet;
  signalroute RInt
   from Marker to Sender with prior_data, norm_data;
  signalroute RConf
    from PSender to env with conf;
  signalroute ROut1
    from PSender to env with packet;
  signalroute ROut2
    from PSender to env with packet;
  signalroute Rout3
    from Receiver to env with packet;
  connect RIn1 and CInt1;
  connect RIn2 and COut1;
```

```
    connect RConf and CInt;
    connect ROut1 and COut1;
    connect ROut2 and COut2;
    process Marker referenced;
    process Sender referenced;
    process PSender referenced;
    process Receiver referenced;
endblock Router;
```

В нем представлены два процесса `Marker` и `PSender` и семь маршрутов: `RIn1`, `RIn2`, `RInt`, `RConf`, `ROut1`, `ROut2` и `Rout3`. Процесс `Marker` соединен с рамкой блока маршрутом `RIn`, по которому поступают сигналы `data` и `control`. По маршруту `RInt` процесс `Marker` передает сигналы `prior_data` и `norm_data`.

Процесс `PSender` соединен с рамкой блока односторонними маршрутами `Rout1`, `ROut2` и `RConf`, которые присоединены к каналам `COut1`, `COut2` и `CInt`, соответственно. По маршрутам `Rout1` и `ROut2` может передаваться сигнал `packet` , по `RConf` — сигнал `conf`.

Процес `Receiver` соединен с рамкой блока односторонними маршрутами `RInt2` и `Rout3`, которые присоеденены к каналам `COut1` и `CInt`, соответственно. По маршруту `RIn2` к процессу поступают сигналы `packet`, по маршруту `Rout3` в окружение блока передаются сигналы `data`. Графическое представление блока `Router` приведено на рис. 6.

Сеть, моделирующая блок `Router`, представлена на рис. 7.

Для порождения экземпляров процессов на этом шаге создаются дополнительные места *Create_id* (где *id* — имя порождаемого процесса). Каждое место *Create_id* является входным для модуля, соответствующего порождаемому процессу с именем *id,* и выходным для модуля, представляющего родительский процесс. Например, если некоторый процесс в своих переходах содержит *k* операторов `CREATE`, но каждый из них имеет вид либо `CREATE A`, либо `CREATE B`, то в сети будет создано два места *Create_A* и *Create_B*, которые будут выходными для модуля, соответствующего этому процессу. Таким образом, можно сказать, что каждое место *Create_id* вместе с входной и выходной дугой моделирует пунктирную стрелку в графическом описании языка SDL, соединяющую описание процесса-родителя с описанием порождаемого процесса.

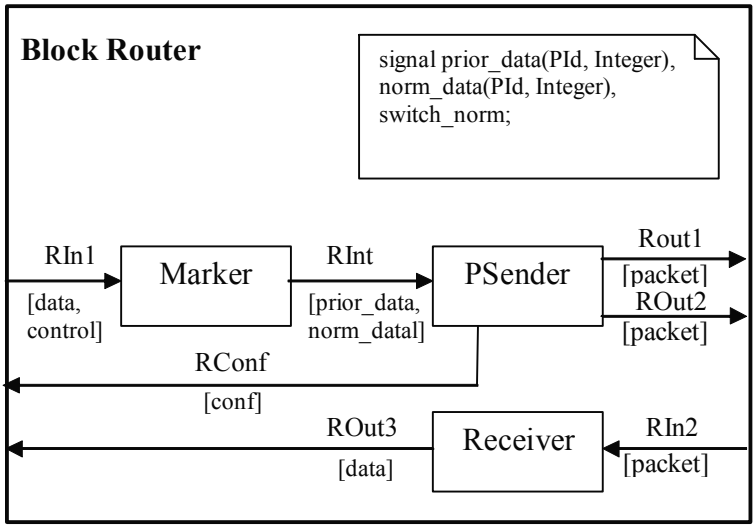


Рис. 6. Описание блока `Router`

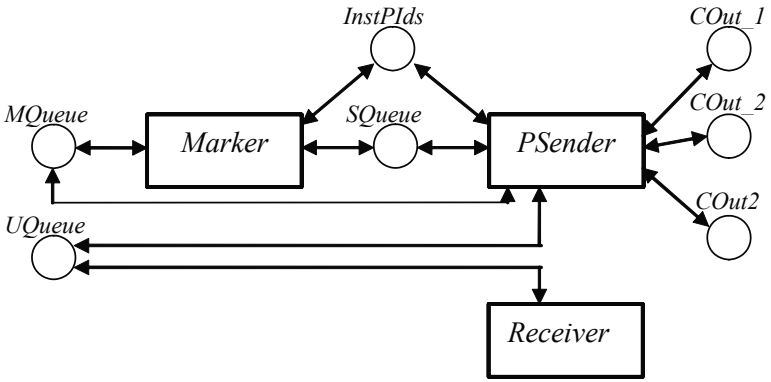


Рис. 7. Сетевое представление блока `Router`

Если процесс не имеет формальных параметров, фишки в месте *Create_id* могут принимать значения из множества цветов

*pair = product integer * integer*,

где значение первого поля есть личный идентификатор создаваемого оператором `CREATE` экземпляра процесса, значение второго поля — личный идентификатор «экземпляра-родителя».

В случае, когда процесс имеет формальные параметры, декларации сети расширяются определениями двух типов. Первый тип, назовём его *Create_params*, представляет собой запись, в которой каждому формальному параметру процесса соответствует поле соответствующего типа. Так как в CPN Tools запрещается определять типы-записи, содержащие единственный элемент, в случае, когда процесс имеет один формальный параметр, в *Create_params* добавляется фиктивное поле *null* целого типа. Второй тип, назовём его *Create_type*, является кортежом. Он содержит три элемента: личный идентификатор создаваемого экземпляра процесса, личный идентификатор «экземпляра-родителя» и формальные параметры (элемент типа *Create_Params*).

Пример. Для процесса `P`, имеющего формальный параметр `resp` целого типа, декларации сети будут расширены следующими определениями:

*colset P_Create_params = record resp:integer * null:integer;*

*colset P_Create_type = product integer * integer * P_Create_params timed;*

Каждой обозреваемой переменной соответствует одно место. Оно является входным и выходным местом как для модуля, представляющего экземпляр процесса, который раскрывает переменную, так и для модулей, представляющих экземпляры процессов, которые обозревают эту переменную. Типом данного места является запись, второе поле которой соответствует сорту данной переменной.

Блок может содержать процессы, которые экспортируют/импортируют значения некоторой переменной другим процессам из этого же блока. Каждой экспортируемой переменной ставится в соответствие одно служебное место — входное и выходное для модулей, соответствующих блокам, которые содержат процессы «экспортер» и «импортер».

3.4. Моделирование процессов

Моделирование порождения и уничтожения экземпляров процессов основано на том, что многоуровневое описание системы в SDL имеет статический шаблон, т. е. число экземпляров процесса может изменяться в про-

цессе функционирования системы, но позиция каждого экземпляра в общей иерархии системы остается неизменной. Статический шаблон моделируется структурой сети, а экземпляры процессов — фишками. Таким образом, различные экземпляры одного процесса моделируются одной и той же сетью, построенной по описанию этого процесса, но различными наборами фишек в местах этой сети.

Поскольку фишки, принадлежащие экземплярам одного и того же процесса, располагаются в одних и тех же местах, возникает необходимость в их дополнительной идентификации. Для этого фишки снабжаются уникальным признаком — персональным идентификатором экземпляра процесса (ПИД). Все фишки, принадлежащие конкретному экземпляру процесса, помечены одним и тем же ПИД.

Все места в строящейся сети, моделирующей процесс, экземпляры которого создаются только во время функционирования системы, изначально не будут иметь никаких фишек. Таким образом, будет заготовлена только «сеть-шаблон», в которой фишки появятся после того, как в сети, соответствующей «процессу-родителю», сработает переход, моделирующий оператор `CREATE`. Подробно это будет описано ниже.

На подстранице, связанной с модулем, соответствующим некоторому описанию процесса**,** отображается внутренняя структура этого процесса. Все сорта, определения сигналов и списков сигналов, описанные в декларативной части процесса, преобразуются в множества цветов и заносятся в декларации сети; при этом при этом имена состояний добавляются в множество цветов *allenums*.

Каждой процедуре, определённой в процессе, соответствует модуль и места *Run* и *Return*. Фишки в месте *Run* соответствуют запросам на вызов соответствующей процедуры, фишки в месте *Return* несут результаты выполнения процедуры.

Каждому SDL-переходу в сети соответствует один модуль, каждой описанной в процессе переменной — одно место. Тип места-переменной есть запись. Второе поле описывается множеством цветов, полученным при отображении сорта этой переменной. Начальная разметка второго поля места-переменной соответствует начальному значению переменной, если оно определено, или нулю, если не определено. Значение первого поля записи есть ПИД экземпляра процесса.

Каждый процесс имеет одну стартовую вершину, за которой следует переход. Этот переход будем называть стартовым. В сети ему соответствует модуль *Start*.

Кроме того, подстраница содержит три служебных места: *Queue*, *State* и *sender*. Место *Queue* представляет порт экземпляров процесса и содержит очереди фишек, соответствующих очередям сигналов, поступающих по всем маршрутам к данным экземплярам процесса. Множество цветов, приписываемое месту *Queue*, есть запись, где первое поле содержит ПИД экземпляра, а второе поле формируется так же, как множество цветов для мест-каналов.

Выполнение SDL-перехода — неделимое действие. Наличие в сети места *State* обеспечивает неделимость выполнения переходов сети, моделирующих SDL-переход. Множество состояний процесса определяет множество цветов второго поля этого места.

Служебное место *sender* соответствует переменной `SENDER`. В момент, когда процесс-получатель воспринимает входной сигнал, его переменной `SENDER` присваивается личный идентификатор того экземпляра процесса, который отправил этот сигнал.

В сети, соответствующей процессу, экземпляры которого создаются при инициализации системы, изначально в служебных местах будет столько фишек, сколько экземпляров процесса создано. Каждая фишка в месте *sender* имеет значение *(n, 0)*, в месте *State* — *(n, e)*, где *n* — есть ПИД конкретного экземпляра.

Входным путем к процессу будем называть путь, начинающийся каналом, идущим из окружения системы к блоку, содержащий этот процесс, и заканчивающийся маршрутом, присоединенным к данному каналу и являющимся входным к этому процессу.

Для каждого входного пути создается служебный переход *Link*, который является входным и выходным для места *Queue* и для места, соответствующего каналу, с которого начинается путь. Срабатывание перехода *Link* моделирует пересылку сигнала из входного канала в порт процесса. При этом из места-канала забирается фишка, представляющая список сигналов, и возвращается обратно фишка, представляющая хвост этого списка. Голова этого списка добавляется в хвост списка фишки в месте *Queue.*

Кроме того, в сети, представляющей процесс, создается служебный переход *Delete*, для которого места *State* и *Queue* будут входными и выходными. Переход *Delete* моделирует потребление первого сигнала из очереди сигналов экземпляра процесса в том случае, если экземпляр, находясь в определенном состоянии, не воспринимает этот сигнал. Спусковая функция перехода *Delete* формируется исходя из состояний, которые имеет процесс, и сигналов, которые воспринимаются процессом.

Если процесс содержит оператор `CREATE`, то на странице этого процесса создается служебное место *offspring*, а на странице, соответствующей порождаемому процессу, создается служебное место *parent*. Фишки в них принимают значения из множества цветов *pair*.

При моделировании процесса, экземпляры которого создаются во время функционирования системы, на странице, соответствующей описанию этого процесса, строится служебный переход *Create*. Все служебные места, места-параметры и места-переменные являются для него выходными. Срабатывание перехода *Create* моделирует создание нового экземпляра и будет описано ниже.

Если процесс содержит конструкцию непрерывного сигнала, то страница, соответствующая процессу, имеет дополнительное служебное место, которое будет входным и выходным для всех модулей, представляющих SDL-переходы в конструкции непрерывного сигнала.

Соединение мест и переходов на этом этапе происходит следующим образом.

- Если некоторая переменная используется в SDL-переходе процесса, то соответствующее данной переменной место будет входным и выходным для модуля, представляющего этот SDL-переход.
- Места *Queue* и *sender* будут входными и выходными для всех модулей, представляющих SDL-переходы; исключение составляют модули, которые входят в конструкцию непрерывного сигнала и содержат разрешающее условие.
- Если в SDL-переходе осуществляется посылка сигнала, то место *InstPIds* будет входным и выходным местом для модуля, который соответствует этому SDL-переходу.
- Если в SDL-переходе процесса осуществляется посылка сигнала, то строятся двунаправленные дуги от соответствующего модуля к местам *Queue* процессов, которые могут получать посылаемый сигнал.
- Место *State* — входное и выходное для каждого модуля и для переходов *Start*.
- Если в SDL-переходе или стартовом переходе процесса вызывается процедура, то место *Run*, расположенное на странице, моделирующей эту процедуру, является выходным для модуля, представляющего этот SDL-переход. Аналогично место *Return* является входным.

Трансляция процедур и функций осуществляется на следующем этапе.

Пример

Проиллюстрируем моделирование процесса PSender, содержащегося в блоке Router (рис. 6).

```
process PSender(1, 1);
  dcl userPId Pid,
      data Integer;
  start;
    output switch_norm to self;
    nextstate prior_send;
  state prior_send;
    input prior_data(userPId, data);
      output packet(userPId, data) via ROut1,
          COut1;
      output conf(data) to userPId;
      nextstate prior_send;
    save norm_data;
    input switch_norm;
      nextstate norm_send;
  state norm_send;
    input prior_data(userPId, data);
      output packet(userPId, data) via ROut1,
          COut1;
lab1:
      output conf(data) to userPId;
      output switch_norm to self;
      nextstate prior_send;
    input norm_data(userPId, data);
      output packet(userPId, data);
      join lab1;
endprocess Sender;
```

Процесс Marker получает сигналы data и control от процессов-клиентов. Сигнал data несет порцию данных. С помощью сигнала control осуществляется переключение режима отправки данных: обычного или приоритетного. Процесс Marker хранит текущий режим для каждого клиента. Переключение режима осуществляется при получении сигнала control со значением PrioritySend или NormalSend.

При получении сигнала data процесс Marker отправляет процессу PSender сигнал prior_data или norm_data, в зависимости от того, включен ли режим приоритетной отправки для клиента, отправившего сигнал data.

Процесс PSender отправляет данные, переданные ему процессом Marker. При этом если в его очереди есть сигналы с приоритетными данными, то они должны отправляться ранее, чем все неприоритетные. Приоритетные сигналы передаются только по основному каналу, а неприоритетные — по любому из имеющихся. После отправки данных в окружение, процессу-отправителю посылается сигнал conf (confirmation), подтверждающий отправку данных.

В данном примере приоритетная отправка реализована с помощью дополнительного сигнала switch_norm и дополнительного состояния. Заметим, что приоритетную отправку можно реализовать с помощью конструкции priority input следующим образом.

Находясь в состоянии prior_send, процесс PSender перебирает все имеющиеся во входном порту сигналы и отправляет данные, соответствующие сигналам prior_data. Все сигналы norm_data сохраняются в очереди. Для прерывания перебора процесс PSender отправляет себе сигнал switch_norm. После того как все сигналы в очереди проверены, происходит переход в состояние norm_send. В этом состоянии отправляется один пакет данных, соответствующий любому из сигналов prior_data и norm_data, а затем производится переход в состояние prior_send.

Процесс Receiver получает сигналы packet, несущие данные из внешнего канала, и передает эти данные процессам-клиентам с помощью сигналов data.

На этом этапе моделирования в сеть, соответствующую процессу PSender, будет добавлено пять модулей, соответствующих переходам процесса, четыре служебных перехода и шесть мест (рис. 8). Чтобы не загромождать рисунок, раскраска мест, начальная разметка мест, условия срабатывания переходов и выражения на дугах опущены.

Множество *all_enums* будет расширено именами состояний процесса PSender:

colset allenums = with e | enum0 | Init | PrioritySend | NormalSend | Close | prior_send | norm_send;

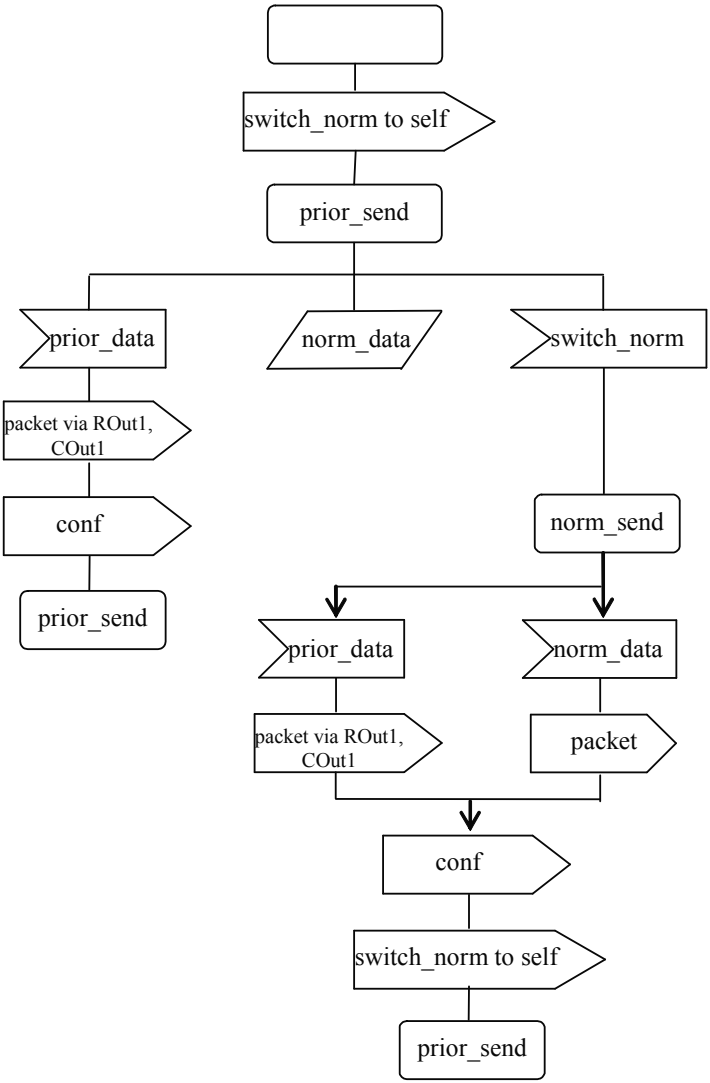


Рис. 8. Описание процесса `Sender`

В декларации сети будут добавлены новые множества цветов мест *State* и *Queue*, соответствующих процессу, и служебные переменные, используемые для связывания значений при срабатывании переходов в подсети:

```
colset P_State_enum = subset allenums with [e, prior_send, norm_send];
colset P_State_type = product integer * P_State_enum timed;
colset P_Queue_unit = record send:integer * sigm:allsigs * pid1:pid
      * integer1:integer;
colset P_Queue_list = list p_Queue_unit;
colset P_Queue_type = product integer * P_Queue_list timed;
colset P_userPId_type = product integer * pidt;
var P_lq, P_lq2: P_Queue_list;
var P_sigval: allsigs;
var P_stateval: P_State_enum;
var P_pid1val: pidt;
var P_integer1val: integer;
var P_parent, P_offspring, P_sender: integer;
var P_pidslist: InstPIds_unit;
var P_userPId: pidt;
var P_data: integer;
```

Подстраница, соответствующая процессу `PSender`, содержит следующие места:

- *sender* соответствует переменной `sender` процесса. Это место имеет тип *pair* и имеет начальную разметку *(pid, 0)*, где *pid* — идентификатор экземпляра процесса `PSender`, создаваемого в момент инициации системы;
- *userPId* и *data* соответствуют переменным процесса. Тип места *userPId — S_userPId_type*, тип места *data — pair*, начальная разметка мест — *(pid, 0)*. Каждое из этих мест соединено двунаправленными дугами с тремя модулями, соответствующими SDL-переходам, в которых параметры сигнала сохраняются в переменных `userPId` и `data`;
- *InstPIds* связано двунаправленными дугами с модулями, соответствующими SDL-переходам, содержащим оператор отправки вида `OUTPUT … TO` <личный идентификатор> …;
- *State* содержит текущее состояние процесса. Оно связано со всеми модулями и переходами на странице за исключением перехода *Link*. Тип места *State — P_State_type*, начальная разметка — *(pid, e);*

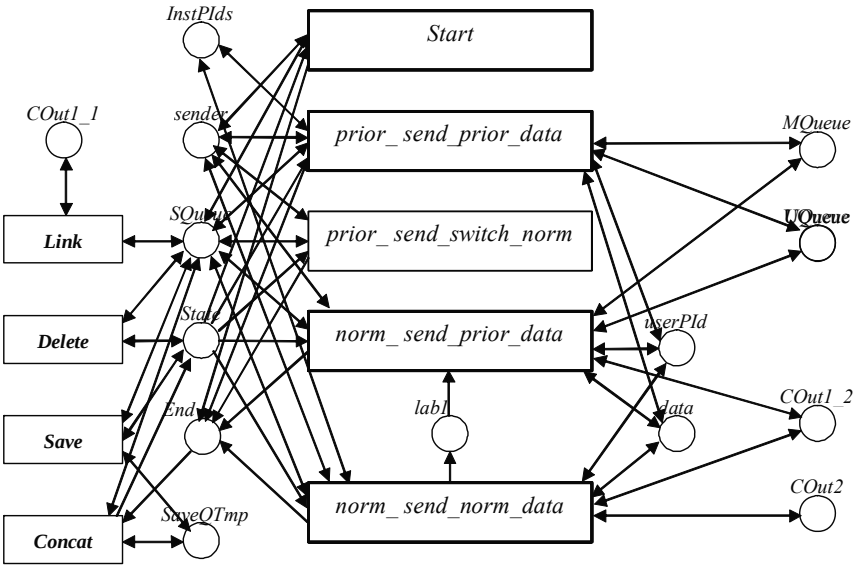


Рис. 9. Сеть, соответствующая процессу `Sender`

- место *SQueue* соответствует входному порту процесса. Это место связано со всеми модулями и переходами на странице. Тип места *SQueue — P_Queue_type*, начальная разметка — *(pid, []);*
- *End* и *SaveQTmp* — места, с помощью которых моделируется сохранение сигналов;
- место *lab1* моделирует передачу управления из одного перехода процесса в другой при выполнении оператора перехода на метку *lab1*. Оно связано входной дугой с модулем *norm_send_norm_data*, соответствующим SDL-переходу с оператором `join`, и выходной дугой — с модулем *norm_send_prior_data*, соответствующим SDL-переходу, содержащим метку. Тип места *lab1 — integer*, начальная разметка — пустая;
- места, соответствующие очередям других процессов в системе — *MQueue* и *SQueue*;

- места *COut1_2* и *COut2*, соответствующие каналам, передающим сигналы в окружение системы.

Служебный переход *Link* моделирует передачу сигнала из канала `COut` в очередь процесса.

Переход *Delete* моделирует удаление из очереди процесса сигналов, не воспринимаемых процессом в текущем состоянии. Процесс `PSender` не воспринимает сигнал `switch_norm`, находясь в состоянии `norm_send`, спусковая функция перехода *Delete* принимает вид

(P_stateval = norm_send andalso (P_sigval = prior_data))

Переходы *Save*, *Concat* и места *End* и *SaveQTmp* моделируют конструкцию `SAVE` и подробно описаны в соответствующем разделе.

Переходы процесса `PSender` на странице, соответствующей процессу, представлены модулями, в дальнейшем они подвергнутся разбиению. Переходу процесса из состояния `prior_send` под воздействием сигнала `switch_norm` будет соответствовать переход *prior_ send_switch_norm.*

Модули, соответствующие SDL-переходам с операторами `OUTPUT`, связаны с местами, соответствующими каналам `COut1` и `COut2` и местами *Queue*, соответствующими очередям других процессов в системе.

Например, переход процесса из состояния `prior_send` под воздействием сигнала `prior_data` содержит операторы отправки

```
OUTPUT packet(userPId, data) via ROut1, COut1;
OUTPUT conf(data) to userPId;
```

Трансляция первого оператора `OUTPUT` приводит к появлению двунаправленной дуги к месту *COut1_2*, соответствующему потоку сигналов, идущих в окружение системы по каналу `COut1`. При трансляции второго оператора `OUTPUT` невозможно определить процесс-получатель сигнала. На данном этапе трансляции модули, соответствующие SDL-переходам, содержащим этот оператор, соединяются с местами, соответствующими очередям всех процессов в системе — *SQueue*, *MQueue* и *UQueue*. Полностью алгоритм трансляции операторов отправки сигналов описан ниже в соответствующем разделе.

3.5. Моделирование перехода

Выполнение некоторых SDL-переходов может моделироваться одним переходом в сети, что происходит в том случае, если SDL-переход не содержит таких действий, как принятие решений, переход на метку, установка или сброс таймера, сохранение сигнала, разрешающего условия и вызовы процедур. Такой переход процесса представляет собой набор операто-

ров присваивания и, возможно, операторов передачи сигналов. При этом ни один из этих операторов не использует переменной, измененной в этом переходе. Такие SDL-переходы будем называть *простыми*, и для них данный этап моделирования станет последним: будут определены спусковые функции и выражения на входных и выходных дугах. Действия, заключенные между описанием входных сигналов и замыкателем перехода, назовем *телом* этого перехода.

Термин «переход» используется и в языке SDL, и в сетях, поэтому при описании моделирования будем использовать приставку N- для обозначения перехода сети в тех случаях, когда значение термина не очевидно из контекста.

В случае простого SDL-перехода на соответствующей ему странице будет присутствовать один N-переход. Состояния, указанные после служебного слова `STATE`, входные сигналы, указанные после служебного слова `INPUT`, образуют спусковую функцию N-перехода. На странице также присутствуют копии всех мест, которые связаны с модулем, представляющим SDL-переход. Входные/выходные дуги повторяют соединение мест-прототипов с модулем. Выражения на дугах определяются оператором `NEXTSTATE` и операторами, составляющими тело SDL-перехода.

Заметим, что при принятых ограничениях динамического поведения системы состояние экземпляра процесса характеризуется собственно его состоянием, содержимым очередей, ассоциированных с маршрутами и портами, состоянием таймеров и значением всех переменных, относящихся к соответствующему экземпляру. При моделировании состояние экземпляра процесса отображается разметкой сети. Срабатывание моделирующего N-перехода возможно при некоторой разметке тогда и только тогда, когда в соответствующем состоянии процесса может выполниться соответствующий SDL-переход.

Рассмотрим переход процесса, описанного на рис. 8, из состояния `prior_send` под воздействием сигнала `switch_norm`. У процесса функционирует один экземпляр. Предположим, что он моделируется фишками, помеченными номером *pid*. В сети этот переход представляется одним переходом. Моделирующая подсеть изображена на рис. 10.

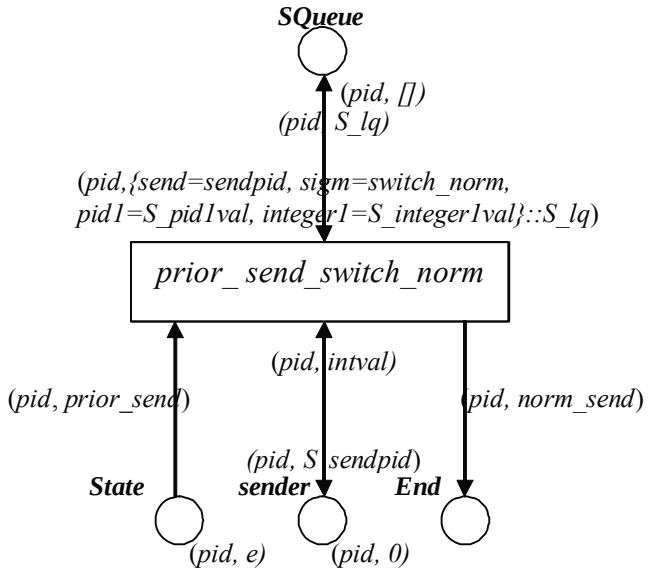


Рис. 10. Сеть, моделирующая переход процесса `Sender`

Переход *prior_send_switch_norm* может сработать, если:

– место *State* содержит фишку, второе поле которой есть *prior_send*, что соответствует состоянию процесса `prior_send`;
– поле *sigm* первого элемента в списке в месте *SQueue* имеет значение *switch_norm*.

Срабатывание перехода *prior_send_switch_norm* заключается в следующем:

– из места *State* забирается фишка со значением *(pid, switch_norm)*;
– в место *End* помещается фишка *(pid, norm_send)*;
– из места *SQueue* забирается фишка со списком, от него отделяется первый элемент, а именно запись вида *{send=sendpid, sigm=switch_norm, pid1=P_pid1val, integer1=P_integer1val}*. Поля *pid1* и *integer1* содержат параметры сигналов `norm_data` и `prior_data`. В место *Queue* возвращается фишка со значением *P_lq* (где *P_lq* — хвост этого списка);
– из места *sender* забирается фишка (*pid, intval*) и возвращается фишка *(pid, sendpid*).

SDL-переходы, которые невозможно представить одним N-переходом, будем называть *сложными*. Для сложных переходов применяется тактика разбиения тела перехода на последовательность фрагментов. Каждый фрагмент рассматривается отдельно, ему сопоставляется подсеть. Поскольку выполнение SDL-перехода происходит последовательно, во всех подсетях можно выделить стартовый и конечный переходы. Подсети соединяются друг с другом с помощью служебных мест *Connect* в том же порядке, что и фрагменты. Фишки в таком месте содержат ПИД экземпляра. Эти места пусты при начальной разметке. Каждая из подсетей представляет одну из стандартных конструкций `DECISION`, `JOIN`, `SET`, `RESET`, `SAVE`, вызов процедуры или один из операторов тела SDL-перехода. Подсеть может состоять из одного перехода или иметь более сложную структуру.

При отображении стандартных конструкций используются библиотечные подсети. Метке в сети соответствует место. Конструкция

`JOIN` <метка>

моделируется переходом и дугой, являющейся выходной для этого перехода и входной для места, соответствующего метке. Последовательность операторов, образующая ветвь конструкции `DECISION`, в свою очередь рассматривается как фрагмент. Если ветвь содержит оператор `NEXTSTATE`, то в сети ему соответствует N-переход *End*, для которого место *State* является выходным. Выражение на дуге, соединяющей этот N-переход с местом *State*, определяется состоянием, указанным после служебного слова `NEXTSTATE`.

Тактика разбиения фрагментов прекращается по достижении фрагмента, представляющего собой последовательность операторов присваивания и, возможно, операторов `OUTPUT`, ни один из которых не использует переменной, ранее измененной в этом же фрагменте. Такой фрагмент представляется в сети одним переходом.

Цепочку подсетей, представляющих сложный SDL-переход, не содержащий конструкции `DECISION` с оператором `NEXTSTATE`, ограничивают два служебных перехода *Begin* и *End*. В противном случае цепочку подсетей ограничивает переход *Begin* и переходы *End*, по одному на каждый оператор `NEXTSTATE`. Таким образом, выполнение сложного SDL-перехода в сети отображается последовательным срабатыванием всех моделирующих его переходов, начиная с первого перехода *Begin* и заканчивая одним из переходов *End*. Состояния, из которых возможен SDL-переход, входные сигналы, указанные после служебного слова `INPUT`, и информа-

ция о том, что экземпляр процесса существует в системе, образуют спусковую функцию перехода *Begin*.

На странице, представляющей SDL-переход, присутствуют место *sender*, которое является входными и выходными для этого перехода, и место *Queue*, являющееся входным для него. Кроме того, на странице присутствует место *State* — входное для перехода *Begin* и выходное для перехода *End*. Фишка из места *State* забирается, как только срабатывает первый служебный переход, и возвращается в него после срабатывания служебного перехода *End*. Наличие в сети места *State* превращает срабатывание нескольких переходов, соответствующих одному и тому же SDL-переходу конкретного экземпляра процесса, в непрерывное действие. В любой момент времени для набора фишек, соответствующих некоторому экземпляру процесса, может сработать не более одного перехода в сети, моделирующего SDL-переходы этого процесса. При этом в сети могут срабатывать переходы с наборами фишек, относящимся к другим экземплярам. Но это не влияет на правильность выполнения всей системы, так как выполнение разных экземпляров одного процесса независимо. Также в процессе выполнения этих переходов сети ничто не мешает выполнению таких переходов, которые моделируют переходы других процессов.

В качестве примера рассмотрим переход процесса `PSender` (рис. 8) из состояния `prior_send` под воздействием сигнала `prior_data`. Этот переход является сложным, моделирующая сеть представлена на рис. 11.

3.7. Моделирование оператора OUTPUT

Рассмотрим граф связей объектов. С его помощью определяются процессы, которым адресуются сигналы, отправленные оператором `OUTPUT`. Это осуществляется с помощью метода волны. Процесс разметки начинается с вершины графа, соответствующей процессу с оператором `OUTPUT`. При этом рассматриваются только те дуги, которые помечены символом * или именем сигнала, указанного в операторе `OUTPUT`. Кроме того, отсекаются пути, не соответствующие содержимому параметра `VIA` и пути вида *блок->маршрут->процесс->маршрут->блок*. Таким образом, сигнал может только «выйти» из родительского блока процесса-отправителя и «войти» в другой блок, но не может затем снова покинуть блок. Результатом работы метода волны является список *достижимых* процессов и каналов, идущих в окружение системы.

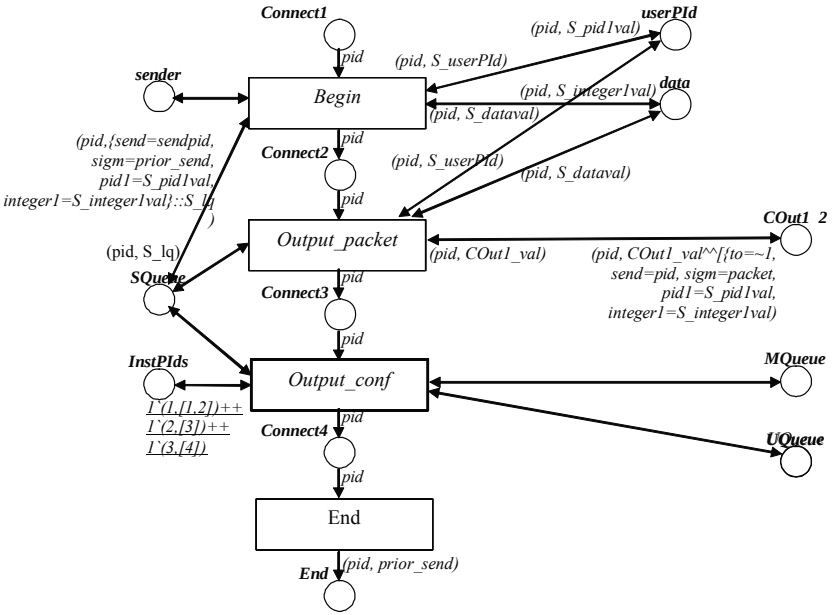


Рис. 11. Сеть, моделирующая переход процесса PSender

Для каждого достижимого процесса в сети создаётся один переход. Также для каждого достижимого канала в сети создаётся один переход. Эти переходы соединяются напрямую с местами, моделирующими очереди процессов-получателей.

Например, если передача сигналов некоторому процессу производится только процессами, находящимися в том же блоке, на странице блока будет присутствовать место-копия очереди процесса, которое соединяется двунаправленными дугами с модулями процесса-получателя и процессов-отправителей. Внутри же модулей это место-копия соединяется с подмодулями, содержащими операторы отправки сигналов и переходами, соответствующими отправке сигналов данному процессу. Кроме того, реализована возможность генерации мест-накопителей сигналов, полученных процессом. Такие места соединяются с переходами, моделирующими оператор отправки.

Если оператор `OUTPUT` не содержит идентификатора экземпляра-получателя, выбор получателя в SDL выполняется в два этапа: на первом этапе определяется один из достижимых процессов-получателей, на втором

этапе — экземпляр выбранного процесса. Если выбирается процесс, не имеющий экземпляров, сигнал теряется. Для моделирования такой ситуции создаётся дополнительный переход *OutputDel*, условием срабатывания которого является наличие пустого списка в месте *InstPIds* для одного из потенциальных процессов-получателей. Если часть процессов-получателей имеет экземпляры, а часть — нет, может сработать как переход *OutputDel*, так и любой из переходов отправки сигнала процессу, имеющему хотя бы один экземпляр. Условие срабатывания такого перехода — наличие в месте *InstPIds* непустого списка для соответствующего процесса. Когда такой переход срабатывает, связывание переменных выбирается случайным образом и получить сигнал может любой из экземпляров. Таким образом моделируется второй этап выбора получателя сигнала.

В качестве получателя оператор `OUTPUT` может содержать

- выражение, задающее ПИД экземпляра-получателя,
- имя процесса-получателя,
- служебное слово `this`.

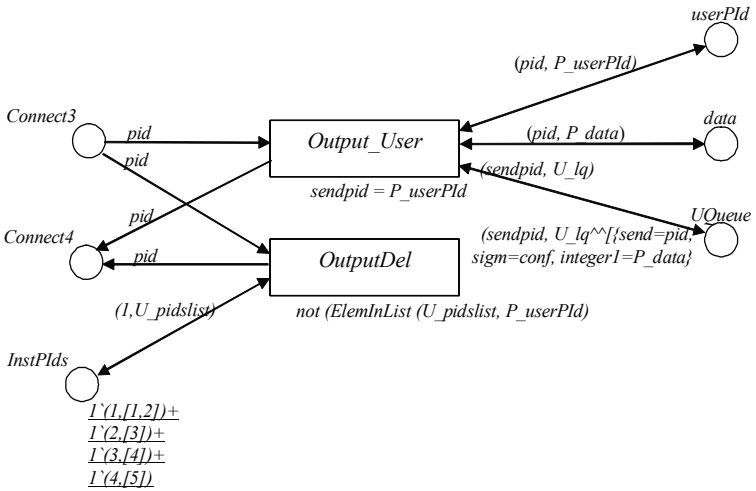


Рис. 12. Моделирование оператора `OUTPUT conf` в процессе `PSender`

В первом случае создается переход для каждого достижимого процесса-получателя, который может получить отправляемый сигнал, и переход для

удаления сигнала *OutputDel*. Однако, для оператора отправки с параметром to `self` всегда создаётся только один переход, помещающий сигнал в очередь экземпляра процесса, выполняющего действие отправки. Для остальных выражений условия срабатывания переходов расширяются и сработать может только переход, соответствующий отправке процессу, имеющему экземпляр с личным идентификатором *pid*, где *pid* равно значению выражения в момент выполнения оператора `OUTPUT`. При срабатывании этого перехода сигнал добавляется в очередь экземпляра с личным идентификатором *pid*. Если экземпляр с таким идентификатором отсутствует или недостижим в момент отправки, сигнал должен быть удалён, это достигается срабатыванием перехода *OutputDel*.

Если указано имя процесса-получателя, сигнал может получить любой из экземпляров указанного процесса. Для реализации этого в сети создаётся два перехода, соответствующих действию отправки сигнала. Первый переход срабатывает, если есть хотя бы один экземпляр процесса-получателя, и помещает сигнал в очередь одного из экземпляров. Второй переход срабатывает при отсутствии экземпляров процесса-получателя и моделирует удаление сигнала.

В третьем случае сигнал может получить любой из экземпляров процесса, выполняющего действие отправки. Это моделируется одним переходом, соединяемым с местом *Queue* текущего процесса.

3.8. Моделирование порождения и уничтожения экземпляров процессов

Оператор `CREATE` в сети представляется тремя переходами: *Generate*, *GenerateNul* и *Create*. Переходы *Generate* и *GenerateNul* располагаются в сети, соответствующей родителю. Переход *Generate* моделирует порождение нового экземпляра процесса, переход *GenerateNul* — холостое срабатывание оператора `CREATE,` когда максимально допустимое количество экземпляров порождаемого процесса уже достигнуто.

Место *Create_id* (где *id* — имя порождаемого процесса) является выходным для перехода *Generate*, а место *NewPId* — входным и выходным. Место *InstPIds* является входным и выходным для переходов *Generate* и *GenerateNul*. Условия срабатывания этих переходов имеют вид *length (pidslist)<n* и *length (pidslist)>=n* соответственно. Здесь *pidslist* — текущий список экземпляров порождаемого процесса, получаемый из места *InstPIds*, а *n* — максимальное количество экземпляров порождаемого процесса.

Переход *Create* располагается в сети порождаемого процесса, и для него служебные места *State*, *sender*, *parent*, места-переменные и места-параметры будут выходными. При срабатывании перехода *Generate* из места *NewPId* забирается фишка со значением номера создаваемого экземпляра процесса, а в очередь в месте *Create_id* добавляется элемент, первое поле которого содержит личный идентификатор порождаемого экземпляра процесса, а второе — личный идентификатор «экземпляра-родителя».

При срабатывании перехода *Create* в сети создается новый экземпляр с номером, равным личному идентификатору порождаемого экземпляра процесса. Это осуществляется путем добавления в каждое место фишки, значение первого поля которой совпадает с ПИД этого экземпляра. Заметим, что срабатывание перехода *Create* произойдет в том случае, если очередь в месте *Create_id* не пуста.

При срабатывании переход *Create*:

- изымает фишку из места *Create_id*;
- изымает фишку из места *NewPId* и возвращает в него значение на единицу больше предыдущего;
- изымает фишку со значением *(n, list)* из места *InstPIds* и возвращает фишку *(n, list^^[pid])* где *n* — идентификатор процесса, экземпляр которого порождается, *pid* — ПИД создаваемого экземпляра процесса, полученный из места *NewPId*;
- помещает фишки в места, соответствующие переменным. Первое поле каждой фишки имеет значение, полученное из *NewPId*. Второе поле фишки соответствует инициализирующему выражению переменной, если оно есть, или «нулевому выражению» в противном случае;
- помещает фишки в места *State*, *Queue*, *sender*, *parent* и *offspring*.

Нулевое выражение определяется как 0 для целых и вещественных типов, символ с кодом 0 — для типа `Character`, false — для `Boolean` и `enum0` — для перечислимых типов. Для массивов и записей нулевое выражение представляет собой массив/запись, заполненный нулевыми выражениями базовых типов.

При срабатывании перехода *Generate*:

- фишка с некоторым значением *p* забирается из места *NewPId* и возвращается в него со значением *(p + 1)*;
- из места *InstPIds* забирается список ПИДов и возвращается список с добавленным элементом *p*;
- из места *offspring* забирается фишка и возвращается в него со значением *(pid, p)*, где *pid* — ПИД экземпляра «процесса-родителя»;

- к очереди в месте *Create_id* добавляется элемент *(p, pid)*, где *p* — ПИД порождаемого экземпляра процесса *id*, *pid* — ПИД экземпляра «процесса-родителя».

При срабатывании перехода *GenerateNul* из места *offspring* забирается фишка со значением последнего созданного потомка и возвращается фишка со значением *0*.

В языке SDL некоторый экземпляр процесса перестает существовать, когда при переходе он входит в состояние `STOP`. Если один экземпляр процесса хочет «уничтожить» своего «брата» или какой-либо экземпляр другого процесса, он посылает сигнал, под влиянием которого последний переходит в состояние `STOP`.

Уничтожение экземпляра процесса осуществляется путем удаления фишек, моделирующих этот экземпляр. Если SDL-переход, приводящий экземпляр процесса с личным идентификатором *pid* в состояние `STOP`, моделируется одним переходом в сети, то при его срабатывании из каждого входного места заберется по одной фишке, принадлежащей экземпляру процесса с номером *pid*. Входными к нему будут места *State*, *Queue*, *sender*, *parent* и *offspring*, а также места-переменные, места-параметры и места, моделирующие таймеры процесса.

При моделировании сложного SDL-перехода места *State*, *Queue*, *sender*, *parent*, *offspring* и места, моделирующие переменные, параметры и таймеры процесса будут входными для служебного перехода *End*. Место *InstPIds* будет входным и выходным для него. При срабатывании перехода *End* из каждого входного места забирается по одной фишке, первое поле которых помечено номером *pid*. Из места *InstPIds* забирается фишка вида *(n, l)*, где *n* — идентификатор уничтожаемого процесса. Из этого списка удаляется ПИД уничтожаемого экземпляра процесса, и список помещается обратно в место *InstPIds*.

3.9. Моделирование процедур

Моделирование процедур языка SDL осуществляется в целом аналогично моделированию процессов SDL. Процедуре соответствует подсеть, содержащая место *State*, моделирующее текущее состояние процедуры, места, соответствующие локальным переменным и набор модулей и/или переходов, моделирующих переходы процедуры из одного состояния в другое.

Экземпляр процедуры создается только в момент вызова процедуры родительским процессом, поэтому место *State* и места, соответствующие переменным процедуры, изначально не содержат фишек. Вызов процедуры и

возврат результата осуществляется через служебные места *Run* и *Return*, располагающиеся на верхней странице сети. После появления фишки в месте *Run* срабатывает переход *CreateCopy*, который помещает в место *State* и места, соответствующие переменным, по фишке, соответствующей новому экземпляру процедуры. Каждая такая фишка содержит в первом поле идентификатор, значение которого равно личному идентификатору экземпляра процесса, вызвавшего процедуру.

На страницах, представляющих модули, соответствующие процедурам, помещаются места, с помощью которых моделируются переменные и таймеры процедур, служебные места *State* и *CallNumber,* а также переход *Start* и переходы, соответствующие SDL-переходам процедуры. Так же как и для процессов, в декларации сети добавляются описания необходимых переменных. Для моделирования порождения экземпляра процедуры при ее вызове и удалении экземпляра при выходе из процедуры создаются служебные переходы *CreateCopy* и *KillCopy* и служебное место *EndProc*.

Во время функционирования сети, после того как сработает переход *Run*, возможно срабатывание либо перехода *Start,* либо перехода *Begin* в модуле *Start*, если стартовый переход процедуры — сложный. Если некоторый SDL-переход в процедуре не заканчивается оператором `NEXTSTATE` или `JOIN`, то должен быть осуществлен возврат из процедуры. Для этого в место *EndProc* передается фишка, после чего должен сработать переход *KillCopy*. При срабатывании последнего забираются фишки, соответствующие данному экземпляру процедуры, из мест, моделирующих переменные процедуры, и места *CallNumber*. В место *Return* помещается фишка с требуемыми выходными значениями и тем же личным идентификатором и кодом возврата, что были переданы при вызове процедуры.

На рис.13 приведен пример процедуры, описываемой на SDL следующим образом

```
procedure P;
  fpar in/out px Integer;

  start;
    task px := (px + 1) mod 8;
endprocedure P;
```

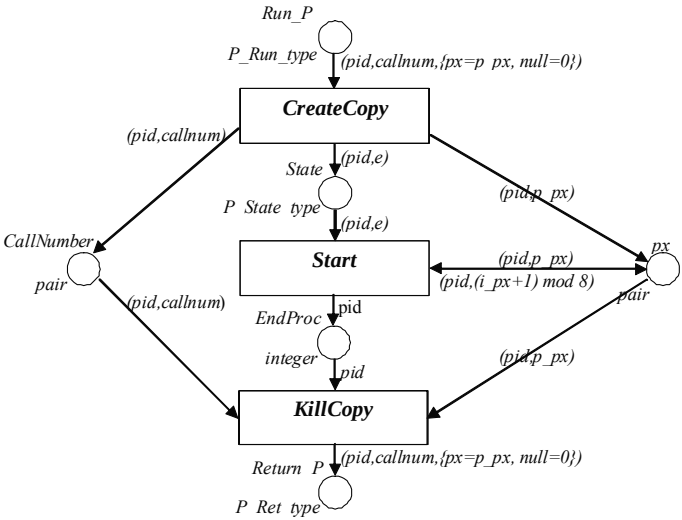


Рис. 13. Пример подсети, соответствующей процедуре

Моделирование оператора вызова процедуры `CALL` осуществляется с помощью двух переходов, назовем их *Call* и *Ret*. Пример подсети, моделирующей оператор вызова, приведен на рис. 14. В этом примере фактическим параметром при вызове процедуры является переменная `X` процесса.

При срабатывании *Call* в место *Run* будет помещена фишка, содержащая личный идентификатор экземпляра процесса, вызывающего процедуру, код возврата и значения фактических параметров. В результате функционирования подсети, моделирующей процедуру, в месте *Return* появится фишка, после этого может сработать переход *Ret*.

При моделировании операторов вызова одной и той же процедуры используется свой код возврата для каждого оператора. После завершения работы процедуры может сработать только один из переходов *Ret*, именно тот, который соответствует сработавшему переходу *Call*. При срабатывании *Ret* результаты выполнения процедуры помещаются в места, моделирующие переменные процесса или вызывающей процедуры.

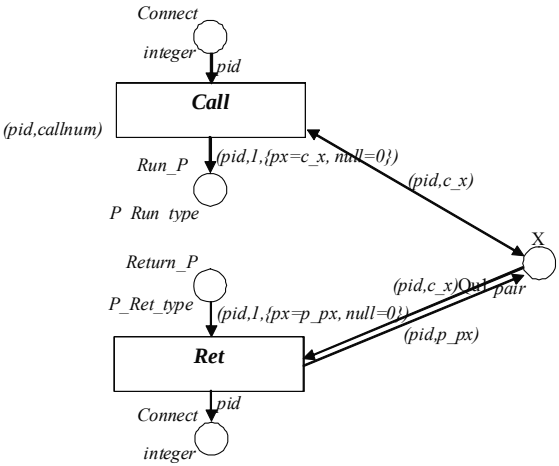


Рис. 14. Пример подсети, соответствующей оператору `CALL`

3.10. Моделирование конструкции SAVE

Рассмотрим моделирование конструкций, связанных со специфической обработкой очереди входных сигналов. При выполнении SDL-перехода из очереди сигналов извлекается тот, который прибыл в нее ранее всех остальных. Но иногда требуется, чтобы процесс, находясь в каком-либо состоянии, мог пропустить вперед сигнал, пришедший после стоящего впереди сигнала. В этом случае используется средство `SAVE` — отсрочка восприятия того сигнала, который находится первым в его очереди. Восприятие этого сигнала откладывается только до перехода процесса в следующее состояние. Если и в этом состоянии восприятие сигнала должно быть отсрочено, то вновь используется конструкция `SAVE`. Если в некотором состоянии отсрочено восприятие нескольких сигналов, стоящих впереди подряд на первых местах в очереди, то средство сохранения сигнала применяется в отношении каждого из них.

Рассмотрим моделирование конструкции сохранения сигналов. На рис. 15 представлена сеть, которая моделирует сохранение сигнала в процессе `PSender`.

В сети создаются дополнительные места *End* и *SaveQTmp*, а также переходы *Save* и *Concat*. Тип места *End* совпадает с типом места *State,* а тип места *SaveQTmp* — с типом места *Queue*. В данном примере существует один экземпляр процесса `PSender`, которому при инициализации модели соответствуют фишки *(3,e), (3,e), (3,[])* в местах *Queue*, *State* и *SaveQTmp* соответственно.

Переход *Save* срабатывает, когда

- в месте *State* находится фишка, соответствующая состоянию `prior_send`,
- в месте *Queue* первый элемент списка содержит сигнал *norm_data*.

При срабатывании перехода *Save* первый элемент списка в месте *Queue*, соответствующий сохраняемому сигналу, помещается в служебное место *SaveQTmp*. Хвост этого списка помещается обратно в место *Queue*. В случае, когда должно быть сохранено несколько сигналов, переход *Save* срабатывает один раз для каждого сигнала, и информация об этих сигналах накапливается в месте *SaveQTmp*.

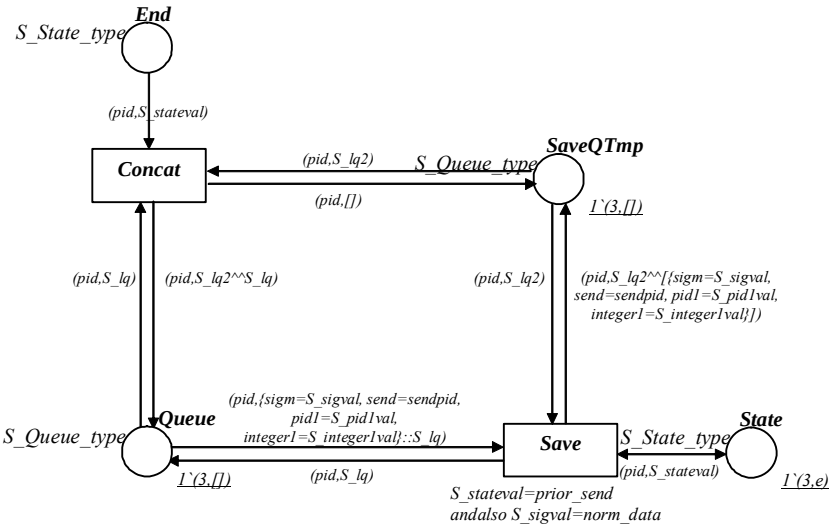


Рис. 15. Сеть, моделирующая конструкцию `SAVE`

Когда в очереди сигналов экземпляра процесса первым оказывается сигнал, который должен быть воспринят, запускается подсеть, моделирующая соответствующий переход SDL. После того как эта сеть отработает, в место *End* помещается фишка вида *(pid, new_state),* где *pid* — идентификатор экземпляра процесса, а *new_state* — новое состояние этого экземпляра. После появления фишки в месте *End* может сработать переход *Concat*, который объединяет списки сигналов, находящихся в местах *Queue* и *SaveQTmp*. В место *SaveQTmp* возвращается фишка, содержащая пустой список. После этого вновь может сработать переход *Save*, если сигнал должен быть сохранен снова. Заметим, что в подсетях, соответствующих процессам, не содержащим конструкции `SAVE`, место *End* отсутствует и фишка *(pid, new_state*) помещается напрямую в место *State*.

3.11. Моделирование временных конструкций

Рассмотрим моделирование SDL-перехода, содержащего оператор установки таймера.

Для отображения таймера `t` в сети создано место *T*, множеством цветов второго поля которого служит множество целых чисел. Начальная разметка этого места — фишка вида *(pid, ~1)* для каждого экземпляра процесса, создаваемого при инициации системы и имеющего личный идентификатор *pid*. Это соответствует тому, что таймеры у экземпляров процесса находятся в неактивном состоянии.

Срабатывание таймера моделируется с помощью служебного перехода *Send*. Этот переход может сработать, когда значение фишки, принадлежащей некоторому экземпляру процесса, имеет вид *(pid, 1)*, и временной штамп фишки равен текущему времени в системе.

Потребление сигнала от таймера каким-либо простым SDL-переходом этого экземпляра процесса моделируется срабатыванием соответствующего N-перехода, которое забирает из очереди запись, соответствующую сигналу от таймера. После этого таймер экземпляра становится неактивным. Потребление сигнала от таймера каким-либо сложным SDL-переходом моделируется срабатыванием первого служебного перехода *Begin* в подсети, моделирующей этот SDL-переход.

Новая установка таймера должна отменять предыдущую. При этой отмене возможны два варианта в зависимости от того, был ли послан в очередь моделируемого процесса сигнал от таймера при предыдущей установке, т. е. достигло ли текущее время в системе значения, заданного преды-

дущей установкой таймера. Если таймер не успел сработать к моменту выполнения новой остановки, никаких дополнительных действий производить не требуется. Если сигнал от таймера уже находится в очереди моделируемого процесса, этот сигнал необходимо удалить. Для этого реализуется функция *DelQSig*, принимающая на вход список сигналов, хранящийся в месте *Queue*.

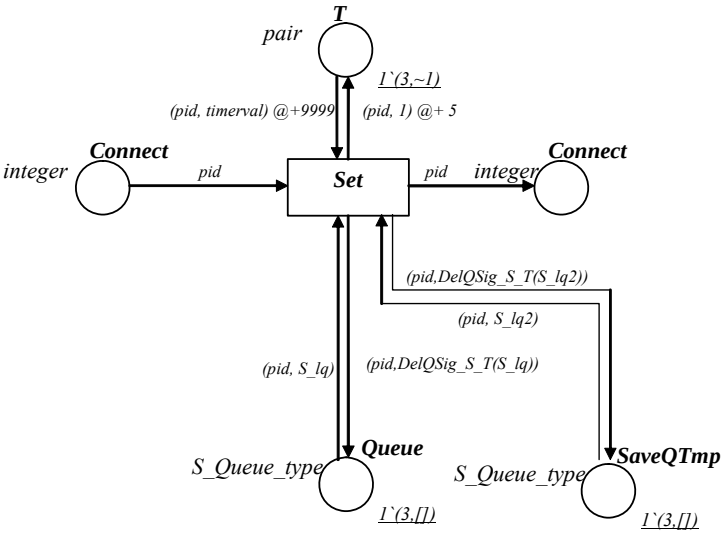


Рис. 16. Сеть, моделирующая установку таймера

Моделирование оператора `RESET` производится аналогичным образом.

3.12.1. Моделирование таймеров с параметрами

Место, моделирующее таймер с параметрами, содержит фишки, первое поле в которых содержит ПИД родительского процесса, а второе — список, содержащий данные о моментах времени, на которые установлен таймер. Назовём этот список *TSets*. Временной штамп такой фишки равен первому моменту времени, в который должен сработать таймер. Если таймер не установлен, то список пуст, а временной штамп фишки меньше либо равен текущему времени в модели. Каждый элемент списка представляет собой запись, содержащую поле для хранения момента времени, на который уста-

новлен таймер и поля, в которых хранятся значения параметров таймера. Записи в списке отсортированы по возрастанию значения первого поля.

Место, моделирующее таймер без параметров, содержит фишки типа *pair*, значение второго поля в которых может принимать значение 1, если таймер может сработать, и –1 в противном случае. Если таймер установлен, временной штамп фишки соответствует моменту времени, в который таймер должен сработать.

Как уже было упомянуто выше, каждому таймеру без параметров соответствует своя функция *DelQSig*. Эта функция принимает в качестве параметра список, назовём его *QSignals*, содержащий информацию о сигналах, находящихся во входном порту одного экземпляра процесса, и удаляет из этого списка запись, соответствующую сигналу от таймера. При этом, каждая такая функция получает имя вида *DelQSig_<дескриптор процесса, в котором определён таймер>_<имя таймера>*, где дескриптор — уникальное в рамках сетевой модели сокращенное имя процесса.

Каждому таймеру с параметрами соответствуют четыре функции: *DelQSig, DelTRec, AddTRec и GetTimeMark*. Функция *DelQSig* принимает в качестве параметров список *QSignals* и набор значений, соответствующих параметрам таймера. Эта функция удаляет из списка сигнал от таймера.

Функция *DelTRec* принимает в качестве параметров список *TSets* и набор значений, соответствующих параметрам таймера. Она удаляет из списка запись, соответствующую установке таймера с заданными значениями параметров.

Функция *AddTRec* принимает в качестве параметров значение *tval*, на которое устанавливается таймер, список *TSets* и набор значений, соответствующих параметрам таймера. Она добавляет в список запись, соответствующую установке таймера с заданными значениями параметров на время `Now+tval`.

Функция *GetTimeMark* принимает в качестве параметров список *TSets* и выдаёт текущую временную задержку. Задержка помещается на дугу от перехода, моделирующего конструкцию *Set/Reset*, к месту, моделирующему таймер. Таким образом, после срабатывания перехода в месте-таймере окажется фишка с временным штампом, равным моменту первого срабатывания таймера. Эта задержка равна $T_{glob} - T_{min}$, где T_{glob} — текущее значение глобальных часов в модели, а T_{min} — значение первого элемента списка *TSets*.

Установка и сброс таймера моделируются одним переходом. Переход соединяется с местами-таймерами, местом *Queue* и местами, моделирующими переменные, значения которых используются в качестве параметров

оператора установки/сброса. На рис. 17 приведён фрагмент сети, соответствующий оператору сброса таймера `T`. Таймер имеет один целочисленный параметр. При установке его значение равно сумме значений двух переменных процесса: `Var1+Var2`. Дескриптор процесса — `P`.

На дугах, соединяющих место *Queue* и переход, моделирующий оператор *Set* или *Reset*, находятся выражения, обеспечивающие удаление ещё не потреблённого сигнала от таймера из входного порта процесса. Список *QSignals* связывается с переменной *<дескриптор процесса>_lq*.

При моделировании таймеров без параметров на дуге, идущей от места-таймера к переходу *Set* или *Reset*, помещается выражение *(pid, timerval*), а на обратной дуге — выражение *(pid, 1)@+dt,* где *dt* — значение задержки срабатывания таймера.

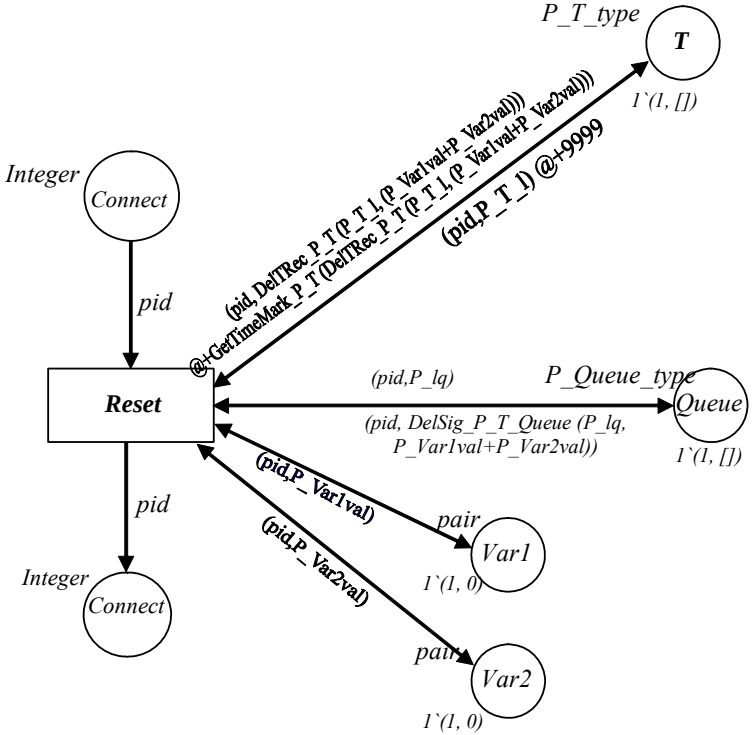


Рис. 17. Сеть, моделирующая установку таймера с параметрами

В случае моделирования таймера с параметрами на дуге, идущей от места таймера к переходу *Set* или *Reset*, помещается выражение для связывания списка *TSets* с переменной, указанной в этом выражении. Для того чтобы переход *Set/Reset* мог сработать до того как фишка в месте таймера станет доступна, выражение на дуге дополняется $@+D_{max}$, где D_{max} — некоторое большое целое число, превышающее максимально возможное значение задержки срабатывания таймера.

На дуге от перехода *Reset* к месту-таймеру помещается выражение, обеспечивающее удаление из списка *TSets* записи, соответствующей предыдущей установке таймера с идентичными значениями параметров. Временная задержка на дуге равна $T_{glob} - T_{min}$, *где* T_{min} — минимальное значение времени в новом списке *TSets*.

На дуге от перехода *Set* к месту-таймеру помещается выражение, обеспечивающее удаление из списка *TSets* записи, и добавление новой записи в *TSets*. Эта запись соответствует новой установке таймера.

Переход *Send*, моделирующий срабатывание таймера без параметров, соединяется двунаправленными дугами с местом-таймером и с местом *Queue*. При срабатывании перехода из места-таймера изымается фишка со значением *(pid, 1)* и возвращается в него со значением *(pid, –1)*. Из места *Queue* изымается фишка *(pid, lq)* и возвращается фишка *(pid, lq^^l),* где *l* – список, состоящий из записи, соответствующий сигналу от таймера.

Переход *Send*, моделирующий срабатывание таймера с параметрами, также соединён с местом-таймером и местом *Queue*. При его срабатывании от списка *TSets*, содержащегося в месте-таймере, отделяется первый элемент, а временной штамп новой фишки вычисляется функцией *GetTimeMark*. В качестве аргумента этой функции используется «хвост» списка *TSets*.

4. ОПТИМИЗАЦИЯ И ОЦЕНКА РАЗМЕРОВ СЕТИ

Наша цель — повышение эффективности моделирования, а следовательно — создание компактных сетей. В этом разделе коснемся вопросов оптизации и визуализации, а также оценок размера результирующей сети.

4.1. Пост-обработка сети

В процессе генерации сетевой модели переходы и места получают имена, передающие их назначение в модели и их связь с объектами исходной спецификации. Так, например, места, соответствующие меткам в специфи-

кации, получают имена вида “Label_имя метки“, а модули сети, соответствующие SDL-переходам, — имена вида “состояние_входной сигнал”.

Система CPN Tools не допускает использования в именах переходов и мест следующих символов: '=', '(', ')', '.', ',', '[', ']', '<', '>', '!', '-', '+', '\", '&', ' '. Имена страниц могут содержать только латинские буквы и цифры. Кроме того, модуль работы с графами достижимости системы CPN Tools не допускает повторения имен мест, переходов и модулей сети.

Чтобы преодолеть указанные ограничения после генерации сети, все запрещенные символы удаляются из имен объектов сети. Далее, к именам всех объектов сети на страницах, соответствующих SDL-процессам, и на страницах более низкого уровня дописывается префикс, являющийся сокращенным именем родительского SDL-процесса или SDL-процедуры. После этого составляется список имен всех объектов сети, и к найденным именам-дубликатам справа приписываются порядковые номера.

Например, в спецификации, описанной в предыдущих разделах, объекты сети, соответствующие процессу `PSender`, получат префикс `P`. Первые переходы цепочек мест и переходов, моделирующих переходы процесса `PSender`, переходы *Begin*, в окончательной сети будут иметь имена *PBegin*, *PBegin2*, *PBegin3* и *PBegin4*.

4.1. Оптимизация сети

Для оптимизации сети просматривается дерево страниц и выполняются следующие действия. На каждой странице ищутся конструкции, которые подлежат оптимизации по трем шаблонам, описанным ниже.

1. Находится служебное место, обозначим его *pl*, с помощью которого соединяются два перехода, соответствующие соседним действиям SDL-перехода. Причем, у этого места должна быть только одна входная и одна выходная дуга, выражения на дугах должно быть *pid*. К таким местам относятся, например, места *Connect*.

Входной переход места *pl* обозначим *t1*, выходной — *t2,* причем переход *t2* не дожен имееть спусковой функции. Для всех мест, соединенных дугами одновременно с переходами *t1* и *t2* должны быть выполнены следующие условия:

а) если есть две дуги между местом и переходом *t1*, одна дуга — от этого места к переходу, другая — от перехода к месту, то выражения на этих дугах одинаковые;

б) если есть дуга от места к *t2*, то выражение на ней — то же, что и на дуге/дугах, соединяющих место и *t1*.

Если конструкция, подходящая под данный шаблон найдена, удаляются дуги, соединяющие *t1* и *t2* с местом *pl*. При этом все дуги перехода *t2* переносятся на переход *t1* с сохранением направления и выражений на них. Если было две дуги, идущие к *t1* и *t2*, обе в одном направлении и от одного места, то при переносе дуга к *t1* замещается дугой к *t2*. Аналогично рассматривается случай, когда две дуги, идущих от *t1* и *t2*, обе в одном направлении, к одному месту. В этом случае в момент переноса дуга от *t1* замещается дугой от *t2*. После этого место *pl* и переход *t2* удаляются.

На рис. 18 представлен пример сокращения фрагмента сети, соответствующего данному шаблону. Переход *Output_x*, моделирующий отправку сигнала с параметром x, соответсвует в шаблоне переходу *t1*, переход *Task_x*, моделирующий увеличение значения переменной x на единицу, в шаблоне соответствует переходу *t2*. Место *Connect*, соединяющее переходы *Output_x* и *Task_x*, соответствует в шаблоне месту *pl*. Выражение на дуге от перехода *Output_x* к месту *Queue* на рисунке сокращено.

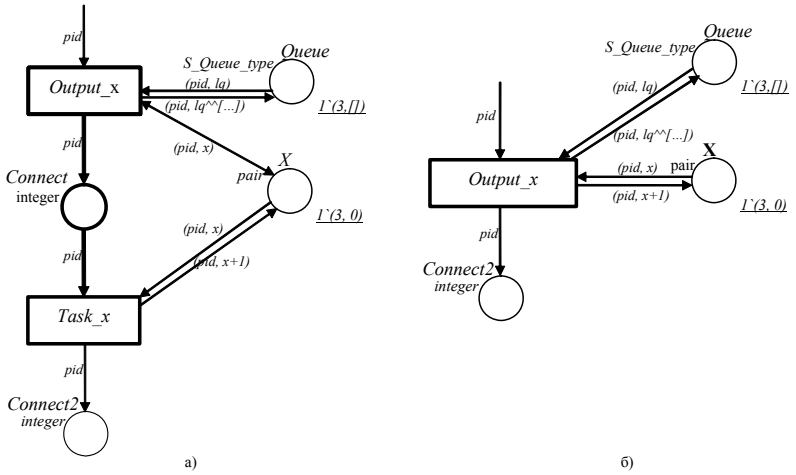


Рис. 18. Пример сокращения сети по шаблону N1.
а) исходный фрагмент сети, б) фрагмент сети после замены

При оптимизации данного фрагмента сети переход *Task_x* и промежуточное место *Connect* будут удалены. Переход *Output_x* будет соединен с местом *Connect2*, с которым был соединен переход *Task_x*. Т

Таким образом, оставшийся переход *Output_x* после сокращения фрагмента сети будет моделировать два действия SDL-перехода — отправку сигнала, несущего текущее значение переменной `x`, и увеличение значения переменной `x` на единицу.

2. Находится переход, обозначим его *t*, у которого нет спусковой функции, но который имеет только по одной входной и выходной дуге. На дугах стоят выражения — *pid*. Обозначим входное для *t* место *pl1*, выходное — *pl*. Причем, место *pl* имеет только одну выходную дугу.

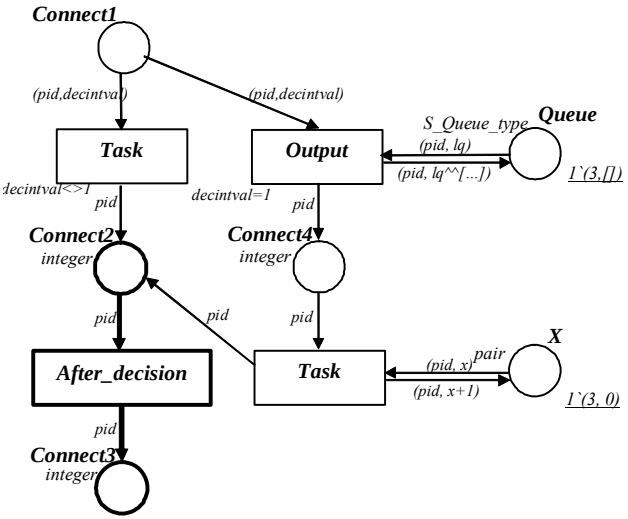
Рассмотрим самую верхнюю страницу, на которой присутствует копия места *pl1*. Если конструкция, подходящая под данный шаблон, найдена, все дуги *pl1* переносятся на *pl* с сохранением направлений и выражений на них. После этого место *pl1* и переход *t* удаляются.

Заметим, что в сетевых моделях, построенных по описанному в главе 3 алгоритму, во-первых, место *pl1* не может иметь более одной выходной дуги. Это следует из того, что более одной выходной дуги, одна из которых содержит пометку *pid*, может иметь только место *Connect,* являющееся выходным для перехода `DECISION`. Но *pl1* не может быть выходным для этого перехода так как не не имеет спусковой функции. Во-вторых, в сети не может быть перехода, соединенного одновременно с местами *pl* и *pl2*.

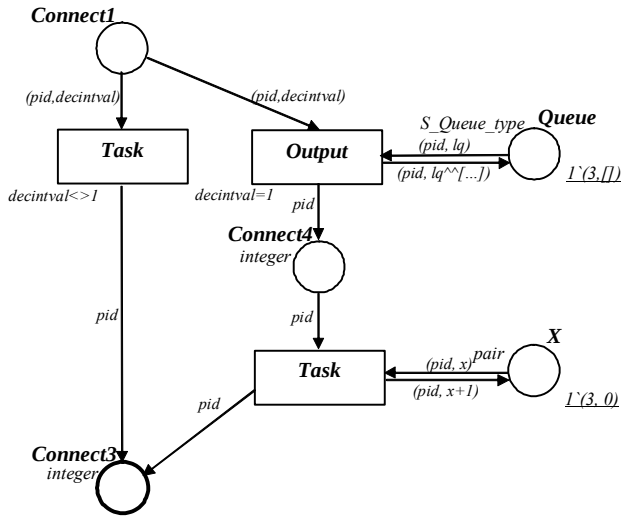
На рис. 19 представлен пример сокращения фрагмента сети, соответствующего данному шаблону. Переход *After_decision*, входящий в конструкцию, моделирующую условный оператор, соответсвует в шаблоне N2 переходу *t*. Место *Connect2* соответствует в шаблоне месту *pl1,* место *Connect3* — месту *pl2*. Выражение на дуге от перехода *Output* к месту *Queue* на рисунке сокращено. При сокращении данного фрагмента сети будут удалены место *Connect2* и переход *After_decision*.

3. Находится место, не имеющее входных дуг и копий на страницах более высокого уровня в иерархии страниц. Если это место имеет пустую начальную разметку, то оно удаляется вместе со своими копиями и выходными дугами.

Оптимизация сети выполняется до тех пор, пока находятся фрагменты, подходящие под описанные выше шаблоны.



а)



б)

Рис. 19. Пример сокращения сети по шаблону N2.
а) исходный фрагмент сети,
б) фрагмент сети после замены

4.2. Размещение объектов на странице

В результате создания внутреннего представления сети определяется количество страниц сети и множеств объектов, которые должны располагаться на каждой странице. Объектам сети приписываются координаты на страницах следующим образом. На каждой странице объекты размещаются в несколько столбцов таким образом, что подавляющее большинство дуг соединяет объекты из соседних столбцов.

Как правило, в самом левом столбце на странице располагаются места, моделирующие каналы, метки и служебные места *Run*, *Return* и *EndProc*. Во второй столбец слева помещаются модули и переходы, соответствующие в спецификации блокам, процессам, процедурам и переходам процессов/процедур, а также служебные переходы *Link*. В третий столбец помещаются места, соответствующие переменным и таймерам процессов и процедур, а также служебные места *Queue*, *State*, *NewPId* и места *Accumulator*, служащие для накопления сигналов, полученных экземпляром процесса. В четвертом столбце располагаются переходы, относящиеся к таймерным конструкциям и конструкциям сохранения сигнала: *Send*, *Trans*, *Concat*, *Save*, *End* и *Qtmp*.

Назначение координат объектов производится независимо для каждой страницы сети. При обработке некоторой страницы объекты разделяются на четыре группы по номеру столбца, в который должен попасть объект. На страницах, соответствующих SDL-переходам, вместо второго столбца строится дерево следующим образом. Сверху располагается переход *Begin*, за ним место *Connect*, затем переход, моделирующий первое действие перехода, место *Connect* и так далее. Если в каком-то месте содержится условный оператор, то цепочки мест и переходов, моделирующие разные ветви условного оператора, располагаются в соседних столбцах. Пример размещения объектов сети, соответствующих переходу процесса `PSender` из состояния `norm_send` под воздействием сигнала `prior_data`, приведен на рис. 20. Процесс `PSender` описан в подразделе 3.4.1.

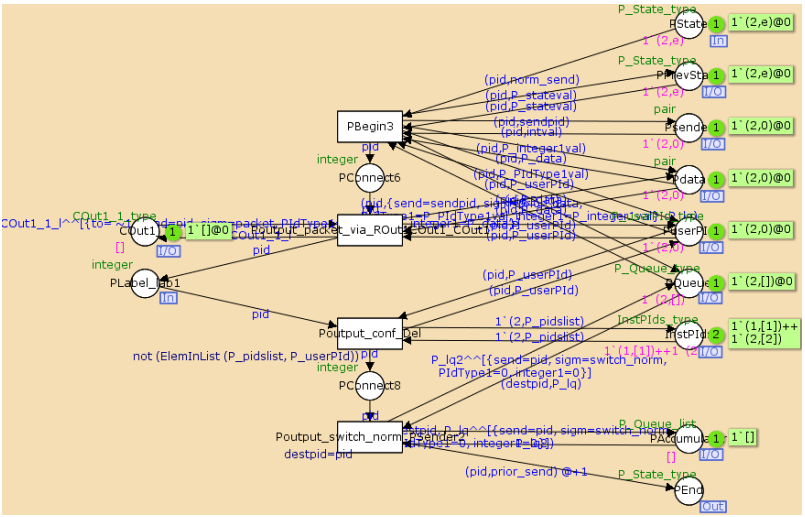


Рис. 20. Пример размещения объектов сети, соответствующих операторам перехода процесса.

Модулям сети, соответствующим блокам, подблокам, процессам и SDL-переходам назначаются координаты *x*, соответствующие второму столбцу слева, и координаты *y,* равные *n·dy, где n = 0, 1, …, N–1* (*N* — количество модулей), а *dy* — некоторый шаг, обеспечивающий достаточную плотность объектов при сохранении удобочитаемости.

По объектам сети, относящимся ко второму столбцу, выстраиваются объекты первого столбца по следующему алгоритму. Для каждого объекта, относящегося к первому столбцу, подсчитывается количество дуг, соединяющих его с объектами из второго столбца. При этом двунаправленная дуга считается за две дуги. Далее, назначаются вертикальные координаты объектов первого столбца в порядке уменьшения количества дуг. Назначаемая координата *y* имеет вид $y_o \cdot dy$, где y_o — целое число. Y_o подбирается так, чтобы минимизировать разницу $y_o \cdot dy$ и суммы координат объектов второго столбца, связанных с данным объектом первого столбца. Если некоторый объект уже имеет выбранное значение координаты *y* равное $y_o \cdot dy$, выполняется поиск следующего значения.

Следующий шаг не применяется к страницам, соответствующим SDL-переходам. Объекты, относящиеся ко второму столбцу, по тому же алго-

ритму выстраиваются по первому столбцу, и далее снова первый столбец — по второму. Таким образом, места каналов, маршрутов располагаются вблизи модулей, для которых они являются входными/выходными. Если таких мест значительно больше, чем модулей, на втором шаге модули будут «разрежены» по вертикали, и на третьем шаге будет выполнена окончательная корректировка позиций мест. Далее по второму столбцу выстраивается третий столбец и по третьему — четвертый.

В результате объекты, связанные дугами с большим количеством других объектов, оказываются в середине страницы. Объекты, связанные всего с одним объектом, оказываются рядом с ним, чаще всего на одной горизонтали. Объекты, не связанные ни с одним объектом на данной странице, оказываются в крайних нижних/верхних позициях или заполняют свободные в столбцах места.

Итоговый результат расположения объектов сети, соответствующих процессу `PSender`, приведен на рис. 21.

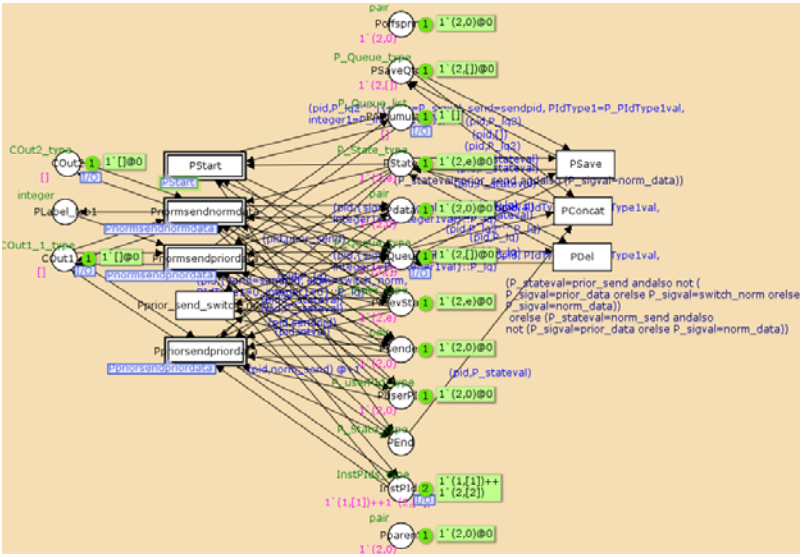


Рис. 21. Пример расположения объектов сети, соответствующих процессу

4.3. Оценка размеров сети

Рассмотрим оценку размера результирующей сетевой модели. Чтобы не учитывать модули и места-копии, оценивается размер эквивалентной неиерархической сети. Рассмотрим процесс, который содержит *var* переменных (в том числе экспортируемых и локальных переменных процедур, определенных в процессе), *par* параметров, *t* таймеров и *N* действий. Если в процессе используются конструкции `SAVE` или `RESET`, то их моделирование требует дополнительных построений, добавляющих один переход и одно место.

В большинстве случаев одно действие перехода процесса моделируется одним переходом и одним соединительным местом. Оператор `TASK`, содержащий N_t присваиваний, моделируется N_t переходами и N_t соединительными местами. Конструкция `DECISION` моделируется не более чем *Nb*2+3* переходами и *Nb+3* местами, где *Nb* — количество ветвей. `CREATE` моделируется одним или двумя переходами и одним соединительным местом. Оператор `OUTPUT`, содержащий отправку единственного сигнала, моделируется не более чем *No+1* переходом и одним соединительным местом, где *No* — количество процессов, которые теоретически могут получить отправляемый сигнал. Оператор, содержащий отправку нескольких сигналов, моделируется как последовательность из нескольких операторов. Оператор `CALL` моделируется двумя переходами и одним соединительным местом.

Описание процедуры моделируется с помощью 2-х служебных переходов и 4-х мест. Для каждой локальной переменной процедуры создается одно место, моделирование действий процедуры производится аналогично действиям процессов.

Назовем *элементарным* действием каждое присваивание, содержащееся в операторе `TASK`, и отправку сигнала, содержащуюся в операторе `OUTPUT`. Одно элементарное действие моделируется не более чем двумя переходом и двумя местами. Заметим, что переходу *Begin*, создаваемому для каждого сложного перехода процесса, не соответствует никакого действия в SDL-спецификации, в свою очередь для оператора `NEXTSTATE` в сети не создается никакого перехода, он моделируется дугой.

Таким образом, сеть, моделирующая описание процесса, будет иметь максимально *TN* переходов и *PN* мест, где

$TN=(2*N+o*(Np+1)+2+t+c_{in})+p*2,$

$PN=(2*N+o+9+var+par+t)+p*4,$

N — количество элементарных действий в процессе и вложенных процедурах; (операторы `TASK`, содержащие *Nt* присваиваний, и операторы `OUTPUT`, содержащие отправку *No* сигналов считаются за *Nt* и *No* действий соответственно),

o — количество операторов отправки сигналов (отправка *No* сигналов считается за *No* операторов),

Np — количество процессов в системе,

p — количество процедур, определенных в процессе.

Моделирующая сеть спецификации состоит из сетей, моделирующих описание процессов, мест-каналов, соединяющих систему с окружающей средой и места *NewPId*.

Таким образом, общая оценка размера сети представляет собой сумму оценок всех процессов, к которой добавляется *1* переход и $c_{in} + c_{out}$ мест, где c_{in} и c_{out} — количество каналов, передающих сигналы из окружения системы и в окружение системы соответственно.

ЗАКЛЮЧЕНИЕ

В настоящей работе предложен алгоритм перевода спецификаций языка SDL в раскрашенные сети Йенсена. В ходе работы алгоритма все переменные отображаются в места. При этом в каждом месте содержится не более чем по одной фишке, относящейся к конкретному экземпляру процесса. Каналы и маршруты представляются местами, в которых в виде списков представлены очереди сообщений. Состояние каждого экземпляра процесса в SDL представляется единственным значением фишки в месте *State*. Служебные места *SaveQTmp*, *Connect, End* также для каждого экземпляра содержат по одной фишке. Порт каждого экземпляра процесса представляется списком в месте *Queue*. Этот факт позволяет не производить полного перебора вариантов связывания при срабатывании переходов сети, что существенно повышает эффективность моделирования.

Обратим внимание на то, что у переходов в сети во входных местах в любой момент времени может быть несколько фишек, соответствующих разным экземплярам процесса, которых достаточно для двух или более срабатываний. Например, при моделировании таймера в месте, соответствующем таймеру, может находиться *n* различных фишек, моделирующих установленные в таймерах времена в *n* различных экземплярах одного процесса. Если окажется, что у каждой из фишек значение временного штампа

совпадает со значением глобальных часов, то переход *send* должен сработать в системе *n* раз.

Процесс оптимизации сети позволяет сократить ее размеры в 1.25−1.65 раза.

Система CPN Tools используется для проектирования и симуляции раскрашенных сетей в лаборатории теоретического программирования ИСИ СО РАН. С ее помощью были проведены исследования протоколов Стеннинга, *i*-протокола, *Ring*, *ATMR* и других. В ходе экспериментов обнаружены новые эффекты поведения протоколов.

За постоянную поддержку, внимание и советы авторы выражают благодарность В. А. Непомнящему и А. В. Быстрову.

СПИСОК ЛИТЕРАТУРЫ

1. Aalto A., Husberg N., Varpaaniemi K. Automatic formal model generation and analysis of SDL // Proc. SDL 2003. — 2003. — P. 285–299. — (Lect. Notes Comput. Sci.; Vol. 2708).
2. Bause F., Kabutz H., Kemper P., Kritzinger P. SDL and Petri net performance analysis of communicating systems. // Proc. IFIP 15th Intern. Symp. on Protocol Specification, Testing and Verification, Warsaw, Poland. — 1995. — P. 259–272.
3. Berthomieu B., Diaz M. Modelling and verification of time dependent systems using time Petri nets // IEEE Transact. on Software Eng. — 1991. — Vol. 17, N. 3. — P. 259–273.
4. Churina T.G., Mashukov M.Yu., Nepomniaschy V.A. Towards verification of SDL specified distributed systems: coloured Petri nets approach // Proc. Workshop on Concurrency, Specification and Programming, Warsaw, 2001. — P. 37–48.
5. Cohen R., Segall A. An efficient reliable ring protocol // IEEE Transact. Communs. — 1991. — Vol. 39, N. 11. — P. 1616–1624.
6. Ratzer A.V. et al. CPN Tools for editing, simulating and analysing coloured Petri nets // Proc. ICATPN 2003. — 2003. — P. 450–462. — (Lect. Notes Comput. Sci.; Vol. 2679).
7. Fisher J., Dimitrov E. Verification of SDL'92 specifications using extended Petri nets // Proc. IFIP 15th Intern. Symp. on Protocol Specification, Testing and Verification, Warsaw, Poland. 1995. P.455–458.
8. Fleischhack H., Grahlmann B. A compositional Petri Net Semantics for SDL // Lect.Notes Comput. Sci. — 1998. — Vol. 1420. — P. 144–164.
9. Grahlmann B. Combining Finite Automata, Parallel Programs and SDL using Petri Nets // Proc. Intern. Conf. TACAS'98. — 1998. — P.102–117. (Lect. Notes Comput. Sci., Vol. 1384).

10. Gammelgaard A., Kristensen J.E. A correctness proof of a translation from SDL to CRL // Proc. of the sixth SDL Forum. Darmstadt, 1993. — P. 205–290.
11. Holzmann G. I. Design and validation of computer protocols. — Englewood Cliffs, NJ: Prentice Hall, 1991.
12. Husberg N., Manner T. Emma: Developing an Industrial Reachability Analyser for SDL // Proc. Intern. Congress on Formal Methods. — 1999. — P. 642–661. — (Lect. Notes Comput. Sci., Vol.1708).
13. Janowski A., Janowski P. Verification of Estelle Specification Using TLA+ // Proc. of 1st. Inter. Workshop on the Formal Description Technique Estelle. — Evry, France, 1998. — P. 109–130.
14. Jensen K. Coloured Petri nets: basic concepts, analysis methods and practical use. — Springer, 1997. — Vol. 1,2,3.
15. Kristensen L.M., Christensen S., Jensen K. The practitioner's guide to coloured Petri nets // Internat. J. on Software Tools for Technology Transfer. — 1998. — Vol. 2, N2. — P. 98–132.
16. V.A. Nepomniaschy, V.S. Argirov, D.M. Beloglazov, A.V. Bystrov,T.G. Churina, M.Yu. Mashukov, R.M. Novikov. Modeling and verification of SDL specified distribu-ted systems using high-level Petri nets. // Proc. Workshop “Concurency, Specification and Programming” (CS&P’2004), Humbold University, Informatics-2004. — Bericht. — P. 100–111. N 170, Berlin.
17. Specification and Description Language (SDL), Recommendation, Z.100, CCITT, 1992.
18. Карабегов А.В., Тер-Микаэлян Т.М. Введение в язык SDL. — М.: Радио и связь, 1993.
19. Непомнящий В.А., Алексеев Г.И., Аргиров В.С., Быстров А.В., Мыльников С.П., Новиков Р.М., Чурина Т.Г. Моделирование и верификация коммуникационных протоколов, представленных на языке SDL, с помощью сетей Петри высокого уровня // Тр. I Всеросс. научной конф. “Методы и средства обработки информации”, (МСО-2003). Москва. МГУ. — 2003. — С. 454–460
20. Чурина Т.Г. Способ построения раскрашенных сетей Петри, моделирующих SDL-системы. — Новосибирск, 1998. — 56 с. — (Препр./ РАН. Сиб. отд-ние. ИСИ; N 56).
21. Чурина Т.Г. Моделирование динамических конструкций языка SDL посредством раскрашенных сетей Петри. — Новосибирск, 1999. — 35 с. — (Препр./ РАН. Сиб. отд-ние. ИСИ; N 71) .

М. Ю. Машуков, Т. Г. Чурина

МОДЕЛИРОВАНИЕ СПЕЦИФИКАЦИЙ ЯЗЫКА SDL С ПОМОЩЬЮ РАСКРАШЕННЫХ СЕТЕЙ ПЕТРИ

Препринт
144

Рукопись поступила в редакцию 06.08.07
Рецензент Е.Н. Боженкова
Редактор З. В. Скок

Подписано в печать 25.09.07
Формат бумаги 60 × 84 1/16 Объем 4.1 уч.-изд.л., 4.4 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. Островского, 55, оф. 02, тел. (383) 214-45-35