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

Т. Г. Чурина, В. С. Аргиров

МОДЕЛИРОВАНИЕ СПЕЦИФИКАЦИЙ ЯЗЫКА SDL С ПОМОЩЬЮ МОДИФИЦИРОВАННЫХ ИВТ-СЕТЕЙ

Препринт
124

Новосибирск 2005

В работе рассматриваются SDL-спецификации распределенных систем с динамическим порождением и уничтожением экземпляров процессов. Для них предложен метод трансляции в модифицированные раскрашенные сети Петри — *иерархические временные типизированные сети (ИВТ-сети),* в которых используются предложенная Мерлином концепция интервального времени, приоритеты и содержатся специальные места, представляющие очереди фишек. Описана реализация метода трансляции и приведены оценки размера сети.

Эта работа частично поддержана грантом РФФИ 03-07-90331.

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

Tatyana G. Churina, Victor S. Argirov

MODELING SDL SPECIFICATIONS USING MODIFIED HTT-NETS

Preprint
124

Novosibirsk 2005

In order to simplify simulation and verification of distributed systems, modified coloured Petri nets called hierarchical timed typed nets (HTT-nets) are introduced. The nets are extended by the priorities, Merlin's time concepts and special places representing queues of tokens. A method for translation from SDL into HTT-nets and its implementation are presented. Complexity bounds of resulted nets which confirm effectiveness of the translation method are given.

ВВЕДЕНИЕ

Анализ, валидация и верификация коммуникационных протоколов — актуальная проблема современного программирования. Один из подходов к решению этой проблемы базируется на использовании таких моделей, как конечные автоматы [5, 11], сети Петри и их обобщения [14], и состоит в моделировании коммуникационных протоколов и верификации полученных моделей. Однако при использовании данного подхода моделирование распределенных систем, как правило, выполняется отдельно для каждого примера, что приводит к необходимости верификации процесса моделирования, а это, в свою очередь, является сложной проблемой для реальных систем.

Автоматический перевод спецификаций в формальные модели, для которых существуют эффективные методы анализа и автоматические средства верификации, решает эту проблему. Известны примеры трансляции спецификаций в конечно-автоматные модели [11], сети Петри [1, 7], 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. Развитие методов трансляции SDL-спецификаций осуществлялось по двум направлениям. При первом используются сети Петри высокого уровня, такие как PrT(predicate-transition)-сети [1, 12] и M-сети [8, 9]. При втором направлении используются специальные классы сетей Петри высокого уровня — SDL-сети, ориентированные на язык SDL [2, 7]. Однако их применение требует разработки специальных методов анализа, неизбежно трудоемких в силу сложности сетей. Как в работе [7], так и в работе [2] описаны сетевые модели, в которых каждому экземпляру процесса соответствует подсеть, и предложены методы построения графа достижимости. Однако для применения этих методов требуются дальнейшие исследования, поскольку графы достижимости весьма громоздки и обычные способы их обработки неэффективны.

Среди различных сетей Петри высокого уровня можно выделить раскрашенные сети Петри (РСП) [14], 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. Следует отметить,

что в общем случае моделирование РСП может приводить к громоздким моделям. Повышение эффективности моделирования — важная проблема. Выделение подклассов РСП и их модификация являются одним из подходов к решению этой проблемы.

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 раскрашенные 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)* [18, 19]. В ИВТ-сетях используются предложенная Мерлином [3] концепция интервального времени, приоритеты и содержатся специальные места, представляющие очереди фишек. В ИВТ-сетях в каждом месте, кроме последних, содержится не более одной фишки.

При моделировании SDL-спецификаций с динамическими конструкциями экземпляры процессов мы отображаем фишками, следовательно, для этих целей ИВТ-сетей недостаточно. Возникла необходимость в модификации сетевой модели. Настоящая работа является логическим продолжением [18, 19] и предлагает способ перевода SDL-спецификаций с динамическим порождением и уничтожением экземпляров процессов в модифицированные ИВТ-сети.

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 ИВТ-сети, в четвертом — приводится описание реализации и оценка размера сети.

Следует отметить, что наши работы [18, 19, 20] велись независимо от работ [8] и [11], в которых экземпляры процессов также представляются фишками, но при этом используются другие сетевые модели.

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 может 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 или выходящие из блока. В разбиении блока для внешних каналов должны быть указаны имена внутренних каналов, которые подсоединены к внешним. Подсоединение 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` из работы [15], текстовое описание которой приведено ниже, графическое. (*рис. 1*).

```
system S;
  signal s1, s2, s3(Integer), s4, s5, s6;
  channel C1
    from B1 to env with s1, s2;
  endchannel C1;
  channel C2
    from B1 to B2 with s4;
    from B2 to B1 with s5, s6;
  endchannel C2;
  channel C3
    from env to B2 with s3;
    from B2 to env with s1;
  endchannel C3;
  block B1 referenced;
  block B2 referenced;
endsystem S;
```

В ней описаны три канала — `c1`, `c2`, `c3` и два блока — `B1` и `B2`. По каналу `c1`, соединяющему окружение с блоком `B1`, от блока могут передаваться сигналы `s1`, `s2` к окружению. По двунаправленному каналу `c3`, соединяющему блок `B2` с окружением, от блока может быть передан сигнал `s1`, а от окружения блок может получить сигнал `s4`. Между блоками `B1` и `B2` имеется двунаправленный канал `c2`. По нему от блока `B1` к блоку `B2` может передаваться сигнал `s3`, который имеет параметр целого сорта, а в обратном направлении — сигналы `s5`, `s6`. Блоки `B1` и `B2` в системе не описаны.

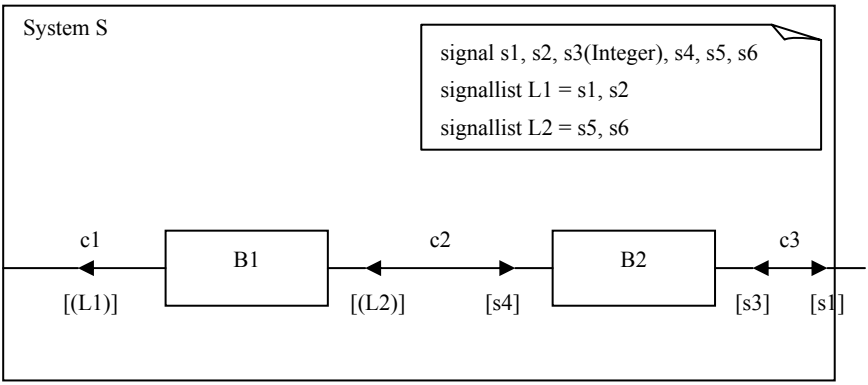

Рис. 1. Описание системы S

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 одновременно в блоке. 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 выполнится пустой переход и они будут удалены 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 трех видов:

OUTPUT <имя сигнала><параметры> TO <личный идентификатор> (1)

OUTPUT <имя сигнала><параметры> (2)

OUTPUT <имя сигнала><параметры> VIA <имя канала> … <имя маршрута> … <имя маршрута> (3)

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

- экземпляру процесса с личным идентификатором, указанным после служебного слова TO;
- экземплярам процессов, связанных с процессом-отправителем маршрутами, которые могут передавать этот сигнал;
- экземплярам процесса, с которым процесс-отправитель связан поименованными в конструкции (после служебного слова VIA) маршрутами.

Заметим, что сигналы всегда несут ПИД экземпляра-отправителя.

Последовательность действий, выполняемых процессом во время перехода, может разветвляться. 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 в данном случае. 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 качестве 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 [13]. 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.

ИВТ-сети являются расширением базовой модели сетей Петри. 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.

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 выделять курсивом.

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. 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. Один из них — *product*. Декларация *color PP = product AA * BB * CC* определяет множество цветов, состоящее из всех троек вида *(a, b, c)*, где *a* — значение из *AA*, *b* — из *BB* и *c* — из *CC*. Такие множества цветов будут использоваться для представления записей в спецификациях.

В ИВТ-сетях допускается тип массив. Кроме того, в сети могут присутствовать места-очереди, способные хранить неограниченное число фишек. Новая фишка в таком месте «помещается в очередь» и остается недоступной, пока из места не будут извлечены все фишки, поступившие до нее. При определении типа мест-очередей в ИВТ-сетях используется описатель *queue of*.

Помимо вышеописанных механизмов в нашей модели определен тип *layer(t)* как множество цветов, состоящее из пар вида *(n, t)*, где *n* — номер

слоя, а t — некоторый цвет. Такие места будем называть *многослойными*. Многослойное место, у которого тип t не есть очередь, может содержать не более одной фишки, принадлежащей определенному слою. Иначе такое место может содержать не более одной очереди в каждом слое. В модели возможно конечное количество слоев.

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*.

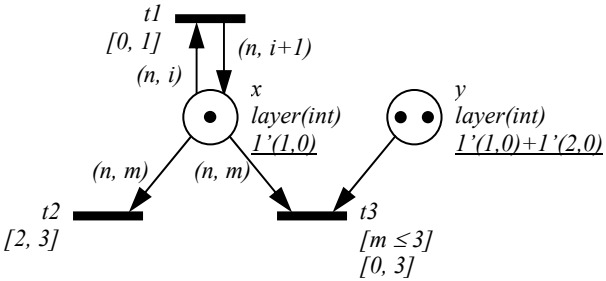

Рис. 2. Пример ИВТ-сети

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. На всех входных

дугах одного перехода в первой позиции кортежа будем использовать одну и ту же переменную.

Функционирование раскрашенной сети отличается от функционирования обыкновенной 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 переменных 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.

Поскольку в ИВТ-сетях на всех входных дугах перехода в первой позиции кортежа используется одна переменная, то при связывании из входных мест выбираются фишки, принадлежащие одному и тому же слою. Заметим также, что возможно многократное срабатывание переходов сети. В любой момент времени переход может иметь во входных местах количество фишек, достаточное для двух или более одновременных срабатываний. Таким образом, переход может сработать несколько раз одновременно, но каждый раз он будет срабатывать с фишками, принадлежащими одному слою.

Для моделирования задержек необходимо явным образом ввести время в модель сетей Петри. Известен ряд временных расширений ординарных и раскрашенных сетей Петри. Для наших целей удобнее определять собственное временное расширение раскрашенных сетей, соединив модели раскрашенных сетей и временных сетей (далее ВСП), предложенных Мерлином [3].

Во ВСП временной механизм связан с переходами: каждому переходу сопоставлена пара неотрицательных чисел d_{min} и d_{max}. Переход, каждое из входных мест которого содержит хотя бы одну фишку, называется *возможным*. В отличие от ординарных сетей Петри не всякий возможный переход имеет право сработать.

Если τ — момент времени, в который переход стал возможен, то его срабатывание произойдет в некоторый момент времени из интервала $[\tau+d_{min}, \tau+d_{max}]$, если условие его возможности не будет нарушено до наступления времени $\tau+d_{max}$ в результате срабатывания другого перехода. Если произошло любое, даже моментальное, нарушение условия возможности перехода, то переход реинициируется, и время задержки его будет заново отсчитываться, начиная с момента восстановления условия его возможности. Возможный переход, имеющий право сработать, т. е. переход, который ожидает не менее d_{min} единиц времени с момента, когда он стал возможен, называется *реализуемым*, а интервал $[d_{min}, d_{max}]$ — *интервалом срабатывания* перехода.

Если переход оставался возможным в течение всего интервала срабатывания и не сработал в течение интервала $[\tau, \tau+d_{max}]$, то срабатывание фор-

сируется и произойдет в момент времени $\tau+d_{max}$. Срабатывание перехода происходит мгновенно.

Таким образом, значение d_{min} определяет минимальное время, в течение которого переход должен оставаться возможным, прежде чем он сможет сработать, а значение d_{max} определяет максимальное время, в течение которого переход может оставаться возможным и не сработать.

Соединение раскрашенных сетей с описанным временным механизмом происходит естественным образом. У переходов сети появляется новый тип пометки — интервал срабатывания. Те же правила, что и раньше, определяют, может ли переход сработать: если во входных местах перехода содержится набор фишек, необходимый для некоторого связывания, то переход может сработать. Само срабатывание происходит с задержкой, значение которой выбирается из интервала срабатывания.

В примере на рис. 2 места *x* и *y* являются многослойными. В месте *x* в слое с номером 1 содержится одна фишка целого типа со значением 0. В месте *y* содержится две фишки, одна в слое с номером 1, другая — с номером 2, каждая фишка имеет значение 0. В выражениях на дугах в первой позиции кортежа указана одна и та же переменная *n*. Таким образом, все переходы могут сработать с фишками, принадлежащими одному и тому же слою в сети. Значения *I = 0, m = 0* определяют связывание. Переходы *t1* и *t2* не имеют спусковых функций, их срабатывание зависит только от временных конструкций. Переход *t3* возможен, если значение фишки в месте x не больше 3. Все переходы возможны при начальной разметке, однако, при этой разметке только переходы *t1* и *t3* могут сработать. Переход *t2* не может сработать, так как должно пройти не менее двух единиц времени до момента, когда он может выполниться, в то время как у перехода *t1* правая граница интервала срабатывания есть 1.

При выполнении перехода *t3* из входных мест *x* и *y* заберутся фишки из слоя с номером 1, после чего каждый переход в сети станет невозможным. Последовательность же срабатываний *t1, t1, t1, t1* приводит к состоянию, в котором оба перехода *t2* и *t3* не могут сработать. В этом случае спусковая функция перехода t3 не выполняется, поэтому этот переход невозможен.

ИВТ-сети имеют приоритеты, таким образом, у переходов раскрашенной сети появляется новый тип пометки — приоритет. Из нескольких возможных переходов срабатывает любой, приоритет которого не меньше, чем у остальных возможных переходов.

ИВТ-сети являются *квазибезопасными* в том смысле, что каждое место может иметь не более одной доступной фишки, принадлежащей определенному слою.

3. ПЕРЕВОД SDL-СПЕЦИФИКАЦИЙ В ИВТ-СЕТИ

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.

Для представления стандартных типов целый, булевский и вещественный используются соответствующие множества цветов: *integer*, *boolean* и *real*. Перечислимые множества цветов сопоставляются перечислимым типам. Для определения структурированных типов данных мы допускаем использование массивов и записей. 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 ReceiverType
  literals
    CR, DT;
endnewtype;
newtype UserDataType
  literals
    A, B, C;
endnewtype;
newtype URType
  struct
    Receiver ReceiverType;
    SenderData UserDataType;
endnewtype;
```

трансформируются в следующие множества цветов:

color ReceiverType = with CR | DT,
color UserDataType = with A | B | C,
*color URType = product ReceiverType * UserDataType.*

Для доступа к полям записи требуется указывать имя переменной соответствующего типа в нужной позиции кортежа: *(Receiver, SenderData)*. Присваивание значения какому-либо полю записи в сети представляется корте-

жем на выходной дуге перехода, где в соответствующей этому полю позиции находится указанное значение. Например, операторам

```
TASK Data!Receiver := DT;
TASK Data!SenderData := A;
```

будет соответствовать кортеж *(DT, A)*.

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.

Кроме того, в каждой сети определено множество цветов, состоящее из одного элемента, который не несет информации:

color Char = with e.

Фишка со значением *e* называется *ординарной* или *бесцветной*. Бесцветные фишки используются в сетях, моделирующих SDL-системы в служебных целях.

3.1. 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. На этом же этапе отображаются каналы. Каждый канал в SDL имеет ассоциированную с ним FIFO-очередь, в которой сохраняются сигналы, полученные через этот канал. Этим же свойством обладают и маршруты. При трансляции SDL-системы каналы, не подвергающиеся разбиению в процессе дальнейшей детализации системы, и маршруты естественно представлять местами специального типа — очередями, а сигналы, сохраняемые в очередях, — фишками. Кроме того, каждая фишка должна содержать информацию о том, какой процесс ее отправил и какому процессу она предназначается. При начальной разметке все места, соответствующие каналам, пусты.

Однонаправленный канал в сети представляется одним местом. Входной канал отображается местом, являющимся входным для модуля, пред-

ставляющего блок, который связан с этим каналом, выходной канал — выходным местом. Двунаправленный канал представляется двумя местами. Одно место моделирует входной поток сигналов, поступающих к блоку, другое — выходной поток сигналов от блока.

Описания всех сигналов, которые могут передаваться по каналам в системе, переходят в декларации сети. При этом создаются множества цветов, которые будут использоваться при моделировании взаимодействий между блоками. Так как выходные сигналы посылаются операторами `OUTPUT` трех видов и всегда несут ПИД экземпляра-отправителя, в каналы передаются сообщения, которые формируются тремя способами, и состоят из следующих элементов:

1) ПИД экземпляра-получателя, либо специальное значение *e*, указывающее, что сообщение предназначается любому экземпляру, либо строка, состоящая из идентификаторов каналов и маршрутов, указанных в конструкции `OUTPUT` после служебного слова `VIA`;
2) ПИД экземпляра-отправителя;
3) сигнал, возможно с параметрами, указанный в операторе `OUTPUT`.

Таким образом, формат сообщения в канале зависит от того, каким оператором сигнал был отправлен в этот канал. Поэтому множество цветов, приписываемое месту-каналу, в общем случае формируется следующим образом:

color Receiver_Pid = union to_Pid: integer + to_all: E + via_str: string
*color Channel_type = product Receiver_Pid * integer * sig_type,*

где *sig_type* — множество цветов, полученное при отображении сигналов, передаваемых по каналу. Множество цветов *Receiver_Pid* позволяет формировать сообщения трех форматов.

Фишка, принимающая значения из множества *Channel_type*, имеет вид *(id, mid, S)*, где *id* — либо личный идентификатор экземпляра процесса-получателя, либо значение *e*, либо строка идентификаторов; *mid* — личный идентификатор экземпляра процесса-отправителя; *S* — сигнал.

Мы рассматриваем спецификации, в которых к одному процессу по всем входным маршрутам поступают сигналы, отправленные операторами `OUTPUT` либо видов (1, 2), либо вида (3). Таким образом, спецификации могут содержать процессы, часть из которых принимает сигналы, отправленные конкретно данному получателю либо любому, а другая часть принимает сигналы, отправленные с помощью конструкции `VIA`. В нашей системе моделирования один процесс не может принимать сигналы, отправленные с конкретной адресацией сигнала и с помощью конструкции `VIA`.

Это ограничение связано с построением спусковых функций и будет подробно описано в разделе трансляции процессов.

Канал, который в дальнейшем подвергнется разбиению, в сети представляется дополнительным модулем и местами. Если канал однонаправленный, то в сети создается пара мест, одно из которых является входным для этого модуля, другое — выходным. Для двунаправленного канала создается две пары мест. Каждая пара вместе с модулем представляет поток сигналов в одном направлении. Эти места ничем не отличаются от мест, которые представляют каналы, не подвергающиеся в дальнейшем разбиению.

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.

Если SDL-спецификация содержит процессы, использующие установку или сброс таймера, то на первой странице создаются переход *add* с интервалом срабатывания *[1, 1]* и межслойное место *now*, которое содержит одну фишку целого значения. В процессе функционирования моделирующей сети значение фишки в месте *now* определяет текущий момент в системе. Копия места *now* также присутствует на каждой странице, отображающей внутреннюю структуру описания блока, содержащего описание процессов с установкой или сбросом таймера. Начальная разметка места *now* — одна фишка значения ноль.

Для генерации персональных идентификаторов экземпляров процессов используется специальное место *PId*. 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.

Пример

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

color Sig1 = with s1 | s2
color Sig22 = with s5 | s6
color Sig = with s3
*color Sig22 = product Sig * Int*
color Sig31 = with s1
color Sig32 = with s4
*color C_1 = product integer * integer * Sig1*
*color C_21 = product integer * integer * Sig21*
*color C_22 = product integer * integer * Sig22*
*color C_31 = product integer * integer * Sig31*
*color C_32 = product integer * integer * Sig32.*

Первое поле любой фишки, принимающей значения из множества цветов *C_1*, *C_21*, *C_22*, *C_31*, *C_32*, полученного при отображении описаний сигналов, содержит личный идентификатор экземпляра-получателя, второе — личный идентификатор экземпляра-отправителя. Изначально все места, порожденные по описаниям каналов, имеют нулевую разметку.

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.

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. Поэтому в сети отсутствуют места, соответствующие этим переменным, место *now* и переход *add.*

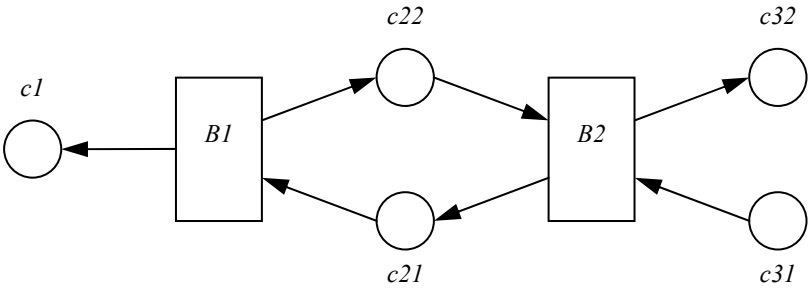

Рис. 3. Сеть для системы S

3.2. 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, назовем *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 соответствует один модуль, каждому внутреннему каналу — одно или два места, в зависимости от того, какой канал задействован — одно- или двунаправленный.

Множество цветов, приписываемое месту, представляющему внутренний канал, является подмножеством цветов места, соответствующего внешнему каналу, к которому присоединен внутренний. Рассмотрим систему `S`, представленную на рис. 1, где блок `B1` имеет подструктуру `B1`, описание которой дано на рис. 4. Каждому подблоку подструктуры `B1` в сети соответствует модуль, имя которого совпадает с именем подблока.

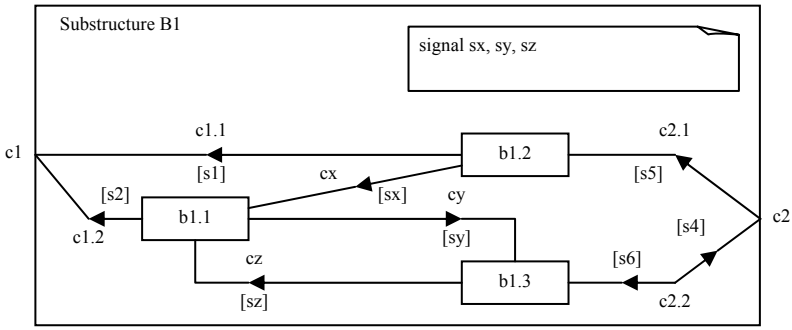

Рис. 4. Описание подструктуры `B1`

На рис. 5 показано, как отображаются в сети внутренние каналы подструктуры `B1`. Место *c1.1* соответствует внутреннему каналу `c1.1` и является выходным для модуля *b1.2*, место *c1.2* соответствует внутреннему каналу `c1.2` и является выходным для модуля *b1.1*, место *c2.1* соответствует внутреннему каналу `c2.1` и является входным для модуля *b1.2*. Каналу `c2.2` в сети соответствует два места: *c2.21* — входное место и *c2.22* — выходное для модуля *b1.3*, моделирующего подблок `b1.3`. Новые однонаправленные каналы `cx`, `cy` и `cz` в сети моделируются местами с такими же именами соответственно.

Каждый сигнал, поступивший по внешнему каналу, может передаваться только по одному внутреннему каналу. По правилам построения иерархической сети копия каждого из мест, представляющих внешний канал, появляется на связанной с модулем странице, которая, в свою очередь, содержит по одному модулю для каждого описания внутреннего подблока или процесса. В строящейся сети места, созданные на предыдущем этапе и соответствующие внешнему каналу, сливаются с местами, соответствующими внутренним каналам, присоединенным к внешнему, следующим образом. Назовем места, созданные на предыдущем этапе и соответствующие внешнему каналу, «старыми местами», а места, созданные на этапе отображения подструктуры и соответствующие внутренним каналам — «новыми местами».

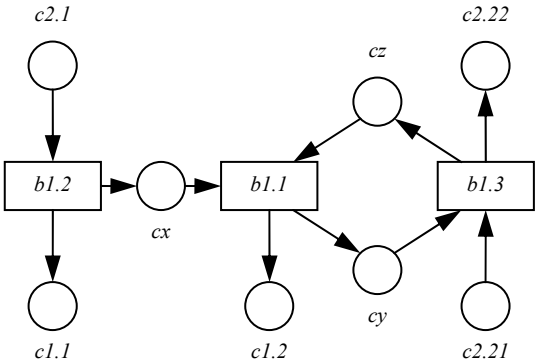

Рис. 5. Сетевое представление подструктуры B1

Таким образом, каждому «старому месту», являющемуся входным для некоторого модуля, представляющего блок, соответствует свой набор «новых мест», которые являются входными местами для модулей, представляющих подблоки данного блока. Аналогично, каждому «старому месту», являющемуся выходным местом некоторого модуля, соответствует свой набор «новых мест», которые являются выходными местами для модулей, представляющих подблоки охватывающего блока.

В нашем примере для «старого места» *c1*, являющегося выходным местом для модуля *B1*, соответствующими «новыми местами» будут *c1.1* и *c1.2*, которые будут выходными местами для модулей *b1.2* и *b1.1* соответственно.

Каждое «старое место», являющееся входным местом для моделирующего блок модуля, сливается с соответствующими ему «новыми местами», которые являются входными местами для модулей, моделирующих подблоки этого блока. Аналогичным образом происходит слияние «старого места» — выходного для модуля, моделирующего блок, с соответствующими ему «новыми местами» — выходными для модулей, моделирующих подблоки этого блока. Заметим, что несколько мест, соответствующих внутренним каналам, могут быть слиты с одним местом, соответствующим внешнему каналу.

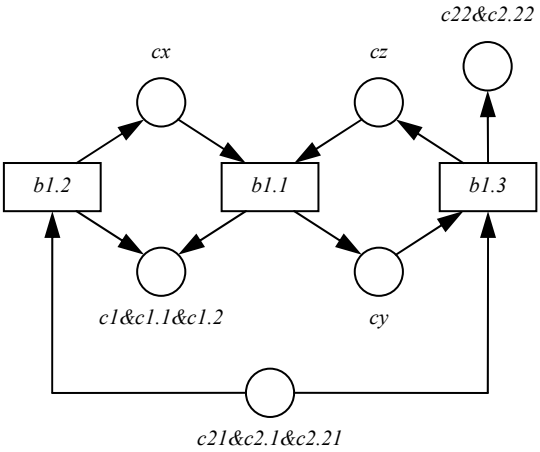

Рис. 6. Результирующая сеть для подструктуры `B1`

На рис. 6 показана страница для подструктуры `B1`, в которой места, соответствующие внутренним каналам, уже слиты с местами, соответствующими внешним каналам. Место *c1* слито с местами *c1.1* и *c1.2*, место *c21* — с местами *c2.1* и *c2.21*, а место *c22* — с местом *c2.22*. В дальнейшем «слитым» местам будем приписывать те же имена, что имеют «старые места» на странице, моделирующей охватывающий блок. Например, вместо имени *c22&c2.22* будет использоваться имя *c22*.

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 множество маршрутов, которые могут передать этот сигнал, и множество экземпляров процессов, которые могут получить сигнал, выбирается произвольным образом.

На рис. 7 показано присоединение маршрутов `m1`, `m2` и `m3` к каналу `c`. Канал может передавать сигналы `s1`, `s2`, `s3` и `s4`. В маршрут `m1` из канала поступают сигналы `s1`, `s2` и `s4`, в `m2` — `s1`, `s2` и `s3`, а в маршрут `m3` — `s3` и `s4`. Присоединение маршрутов с процессами на рис. 8 не показано.

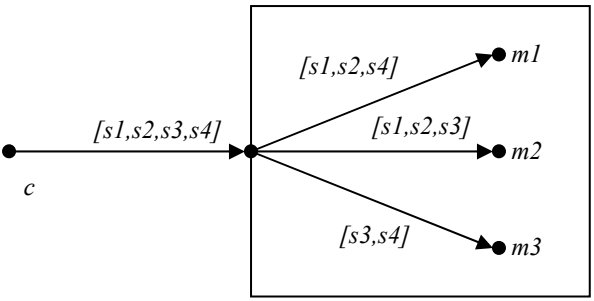

Рис. 7. Присоединение маршрутов `m1`, `m2`, `m3` к каналу `c`

С целью оптимизации сети предложенный алгоритм поддерживает три способа распределения сигналов. При первом способе сигнал из входного канала поступает только в один присоединенный маршрут. Например, сигнал `s1` из канала `c` будет передан либо в маршрут `m1`, либо в `m2`. При реализации этого способа место-канал сливается с местами, представляющими подсоединенные маршруты точно так же, как сливалось «старое место» с «новыми местами» при моделировании подструктуры блока. Таким образом, точке присоединения будет соответствовать одно место *c&m1&m2&m3*. При моделировании этот способ используется в том случае, если все сигналы, поступающие в точку присоединения, отправляются операторами `OUTPUT` вида (1), т. е. с конкретной адресацией сигналов.

При втором способе распределения сигнал из входного канала поступает в каждый присоединенный маршрут, по которому этот сигнал может передаваться. Этот способ используется в том случае, если некоторые сиг-

налы, попадающие в точку присоединения, отправлены операторами OUTPUT вида (2), т. е. сигналы адресуются всем процессам.

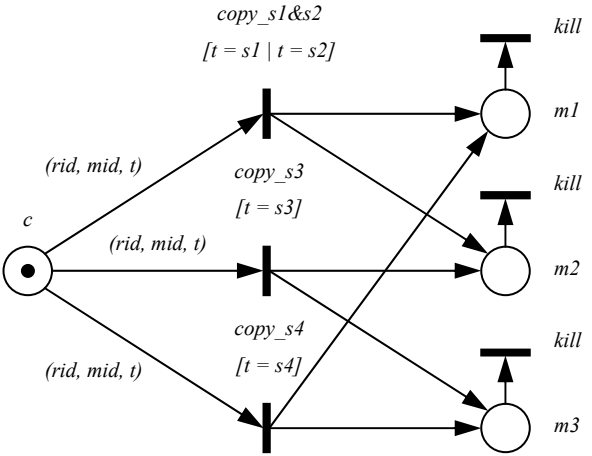

Рис. 8. Отображение точки присоединения маршрутов к каналу

При реализации второго способа в строящейся сети для каждого сигнала, попадающего в моделируемую точку присоединения, создается один служебный переход, назовем его *copy_id*, где *id* — имя сигнала. Место, моделирующее входной канал, является входным для перехода *copy_id*. Места, моделирующие маршруты, которые могут передавать сигнал *id* и являются входными местами для модулей сети, представляющих процессы в этом блоке, будут выходными местами для перехода *copy_id*.

Спусковая функция каждого перехода *copy_id* определяется сигналом *id*. С целью оптимизации в некоторых случаях для нескольких различных сигналов, попадающих в точку присоединения, в сети создается только один переход *copy_id*. Это возможно тогда, когда несколько различных сигналов из входного канала поступают в одни и те же присоединенные маршруты. В примере на рис. 7 таковыми являются сигналы `s1` и `s2`, моделирующая сеть показана на рис. 8. Так как сигналы `s1` и `s2` передаются в одни и те же маршруты, для них в сети создан один переход *copy_s1&s2*.

Третий способ моделирует доставку сигнала из входного канала всем экземплярам процесса, связанного с этим каналом. Этот способ используется в том случае, если процесс имеет несколько экземпляров и если в точку

присоединения попадает хотя бы один сигнал, адресованный всем процессам конструкцией

```
OUTPUT s VIA …,
```

где `s` — имя сигнала. Реализацию этого способа мы опишем в разделе моделирования доставки сигнала всем экземплярам процесса.

Кроме того, на этом этапе моделирования при использовании первого способа отображения точки присоединения в сети создается служебный переход *kill*. Место-канал для этого перехода будет входным. При втором способе отображения для каждого места-маршрута, являющегося входным местом для какого-либо модуля, в сети также создается служебный переход *kill*. Место-маршрут является входным для него. Роль этих переходов будет описана ниже в пункте моделирования процессов.

Для порождения экземпляров процессов на этом шаге создаются дополнительные места *cr_id* (где *id* — имя порождаемого процесса). Каждое место *cr_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 *cr_A* и *cr_B*, которые будут выходными для модуля, соответствующего этому процессу. Таким образом, можно сказать, что каждое место *cr_id* вместе с входной и выходной дугой моделирует пунктирную стрелку в графическом описании языка SDL, соединяющую описание процесса-родителя с описанием порождаемого процесса.

Фишки в месте *cr_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».

На этом же этапе каждому формальному параметру процесса ставится в соответствие одно место — входное и выходное для модуля, представляющего описание процесса, в котором описан параметр. В качестве имени места используется идентификатор соответствующего параметра. Множеством возможных цветов для этого места становится *слой*, где второе поле есть множество цветов, полученное при отображении сорта данного параметра.

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 также является тип *слой*, где второе поле 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».

3.3. 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 процессе 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 и относятся к одному и тому же слою.

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. Множество состояний процесса определяет множество цветов, которое также заносится в декларации сети.

Каждому SDL-переходу в сети соответствует один модуль, каждой описанной в процессе переменной — одно место. Тип места-переменной есть *слой*, где второе поле есть множество цветов, полученное при отображении сорта этой переменной. Начальная разметка второго поля места-переменной соответствует начальному значению переменной, если оно определено, или нулю, если не определено.

Кроме того, подстраница содержит четыре служебных места: *queue*, *self*, *State*, *sender*. Каждое из них является многослойным. 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 *layer*, где второе поле формируется так же, как множество цветов для мест-каналов. При начальной разметке места, соответствующие портам экземпляров процессов, пусты.

Служебное место *self* соответствует переменной `SELF` в SDL, начальная разметка второго поля — номер, который присваивается экземпляру процесса после создания сети.

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 *self — (n, 1)*, в месте *State — (n, e)*, где *n* — номер слоя, который есть не что иное, как ПИД конкретного экземпляра.

Место *queue* соединяется с местами, соответствующими входным маршрутам процесса, посредством служебных переходов *link*. Каждому месту-маршруту соответствует один переход *link*, для которого это место будет входным, а место *queue* — выходным. Срабатывание перехода *link* моделирует пересылку сигнала из соответствующего входного маршрута в порт процесса. При этом из места-маршрута забирается первая из очереди фиш-

ка *(id, mid, sig)*, (где *id* — ПИД экземпляра-получателя и *mid* — ПИД экземпляра-отправителя, *sig* — сигнал) и добавляется в место *queue* в очередь в слое с номером *id*.

При моделировании процессов, воспринимающих сигналы, отправленные с помощью конструкции `VIA`, спусковую функцию перехода *link* образуют строки, состоящие из идентификаторов каналов и маршрутов и представляющие *маршрутизацию* (пути), которая связывает процесс с другими процессами.

Если в порт некоторого процесса поступит сигнал, адресованный экземплярам не этого процесса, он из очереди сигналов удаляется. Удаление осуществляется переходами *kill*, созданными на предыдущем этапе. Для этой цели переходам *link* устанавливается приоритет. Если точка присоединения маршрутов к каналу отображалась первым способом, то переход *kill* может сработать только в том случае, если не может сработать ни один из переходов *link*, соединенных с местом, моделирующим входной канал. Если же точка присоединения отображалась вторым способом, то переход *kill* может сработать только в том случае, когда не может сработать переход *link*, связанный с местом-каналом.

При моделировании процессов, к которым поступают сигналы, отправленные с помощью конструкции `VIA`, спусковая функция перехода *kill*, так же как и переходов *link*, создается на основе маршрутизации. Срабатывание перехода *kill* моделирует удаление из маршрута сообщения, первое поле которого содержит строку идентификаторов, не имеющую отношения ни к одной маршрутизации процесса.

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*.

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.

Каждый процесс имеет одну стартовую вершину, за которой следует переход. Этот переход будем называть стартовым. В сети ему соответствует переход *start*. Место *State* является для него входным и выходным. Спусковая функция перехода *start* позволяет ему сработать, если значение второго поля фишки в месте *State* равно *e*. Выражение на выходной дуге перехода *start*, соединяющей этот переход с местом *State*, определяется состоянием, следующим в процессе непосредственно за стартовым переходом.

Соединение мест и переходов на этом этапе происходит следующим образом.

- Если некоторая переменная используется в переходе процесса, то соответствующее данной переменной место будет входным и выходным для модуля, представляющего этот SDL-переход.
- Если некоторая переменная используется в стартовом переходе процесса, то соответствующее место будет входным и выходным для перехода *start*.
- Места *queue*, *self* 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 *self* будет входным и выходным местом для модуля, который соответствует этому SDL-переходу.
- Если стартовый переход процесса осуществляет посылку сигнала, то место *self* будет входным и выходным местом для перехода *start*.
- Если в SDL-переходе процесса осуществляется посылка сигнала, то проводится дуга от соответствующего модуля к месту, моделирующему выходной маршрут процесса.
- Если стартовый переход процесса осуществляет посылку сигнала, то проводится дуга от перехода *start* к месту, соответствующему выходному маршруту этого процесса.
- Место *State* — входное и выходное для каждого модуля и для переходов *start* и *delete*.
- Если в SDL-переходе осуществляется установка и/или сброс таймера, то место *now* будет входным и выходным местом для модуля, который соответствует этому SDL-переходу.

Трансляция процедур и функций осуществляется на следующем этапе.

Пример

Проиллюстрируем моделирование процесса `Init`, содержащегося в блоке `B2` (рис. 9). Пусть этот процесс соединен с рамкой блока двусторонним маршрутом `m1`, который присоединен к каналу `c2`. По нему может передаваться сигнал `s3` из окружения блока к процессу, а в обратном направлении — сигналы `s5` и `s6`. Пусть также процесс `Init` соединен с рамкой блока двусторонним маршрутом `m2`, который присоединен к каналу `c3`. По этому маршруту может передаваться сигнал `s1` от процесса к окружению, а в обратном направлении — сигнал `s4`. Текстовое и графическое описания процесса приведены на рис. 10.

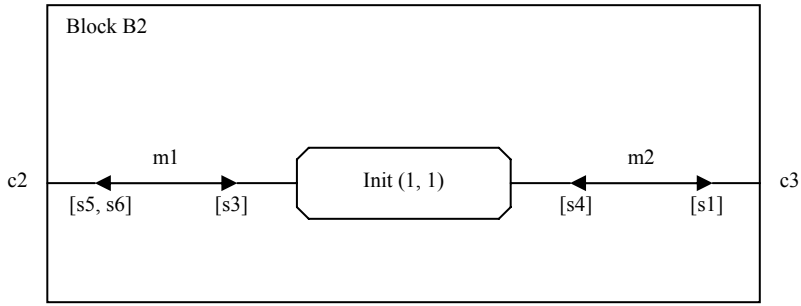

Рис. 9. Описание блока `B2`

Функционирование процесса `Init` происходит следующим образом. Совершив стартовый переход, процесс входит в состояние `disconnected`, из которого под воздействием входных сигналов `s3` или `s4` возможен один из двух переходов. Если первым в очереди процесса находится сигнал `s4`, то совершается первый переход процесса, который воспринимает этот сигнал и посылает в выходной маршрут `m2` сигнал `s1`. Если же первым в очереди сигналов находится сигнал `s3`, то совершается второй переход процесса, который воспринимает этот сигнал, переменной `counter` присваивает значение параметра входного сигнала, посылает в выходной маршрут `m1` сигналы `s5` и `s6`, устанавливает в таймере `t` время `NOW+p` и переходит в состояние `wait`.

```
process Init(1, 1);
  dcl
    counter, x Integer;
  start;
    nextstate disconnected;
  state disconnected;
  (* Первый переход *)
    input s4;
      output s1;
      nextstate disconnected;
  (* Второй переход *)
    input s3(x);
      task counter := x;
      output s5;
      output s6;
      set(now+p, t);
      nextstate wait;
endprocess Init;
```

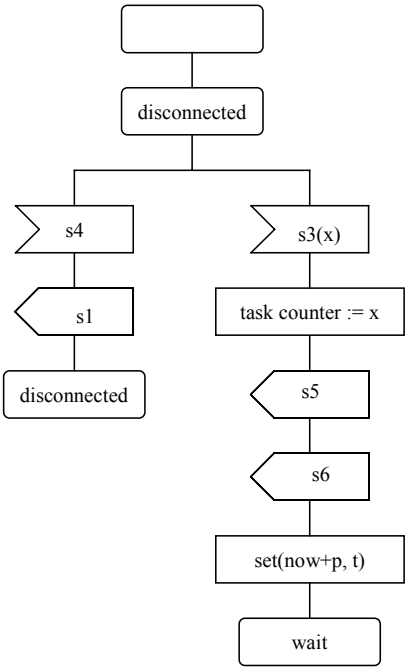

Рис. 10. Описание процесса `Init`

На этом этапе моделирования первому SDL-переходу процесса будет соответствовать модуль *trans1*, второму — *trans2*. Декларации сети будут дополнены новыми множествами цветов:

color dis_con = with disconnected

color wait = with wait

color States_Ini = e | dis_con | wait.

На подстранице, представляющей процесс `Init`, изображена моделирующая сеть (рис. 11). Все фишки этой сети относятся к одному слою, т. е. в первых полях каждой фишки содержится один и тот же номер. В сети содержатся четыре служебных места: *State*, *queue*, *self*, *sender*, место *counter*, представляющее переменную `counter`, а также места *c21&m11*, *c22&m12*, *c31&m21* и *c32&m22*. Места *c32&m22* и *c22&m12* являются выходными соответственно для модулей *trans1* и *trans2* и получены в результате слияния мест-каналов и мест-маршрутов. Места *c21&m11* и *c31&m21*

соединены с местом *queue* переходами *link*. Второе поле фишки в месте *State* может иметь цвет из множества цветов *States_Ini*.

Рассмотрим соединение мест в этой сети с модулем *trans2*. Место *queue* является входным, а места *State*, *self* и *sender* — входными и выходными для него. Поскольку второй переход процесса посылает сигнал в выходной маршрут `m1`, то место *c22&m12* — выходное для модуля *trans2*. После срабатывания перехода *start* в месте *State* появится фишка со значением второго поля, равным *dis_con*.

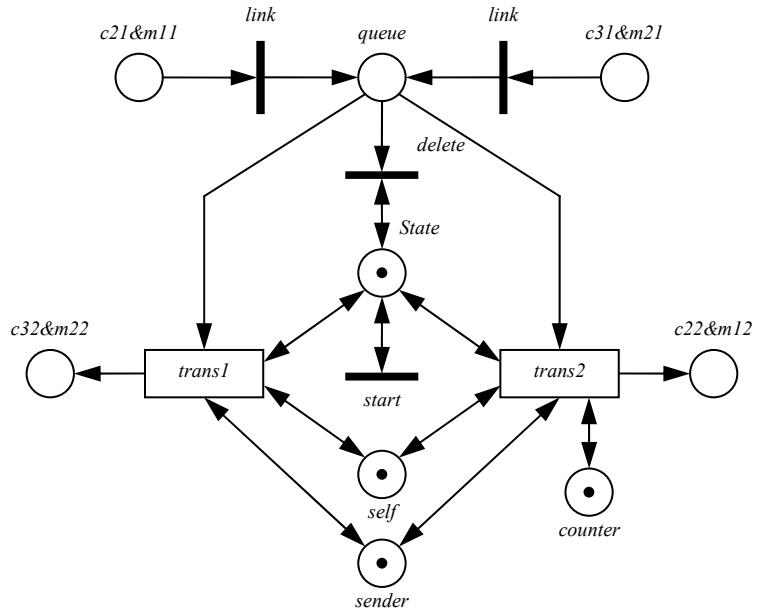

Рис. 11. Сеть, моделирующая процесс `Init`

Спусковая функция перехода *delete* делает возможным его срабатывание, если

- место *State* содержит фишку, второе поле которой имеет цвет *dis_con*, что соответствует состоянию процесса `disconnected`;
- третье поле первой фишки в месте *queue* имеет значение, не совпадающее с цветами *s3* и *s4*.

Срабатывание перехода *delete* моделирует изъятие из верхушки очереди сигнала, который не обрабатывается процессом, находящимся в состоянии `disconnected`.

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 слоя и будет описано ниже.

3.4. 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 операторов 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 используются 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, представляющем 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 пор-

тами, состоянием таймеров и значением всех переменных, относящихся к соответствующему слою сети. При моделировании состояние экземпляра процесса отображается разметкой сети на этом слое. Срабатывание моделирующего N-перехода возможно при некоторой разметке тогда и только тогда, когда в соответствующем состоянии процесса может выполниться соответствующий SDL-переход.

Рассмотрим первый переход процесса, описанного на рис. 10. У процесса функционирует один экземпляр. Предположим, что он моделируется фишками, относящимися к слою с номером *n*. В сети этот переход представляется одним N-переходом. На предыдущем этапе ему был поставлен в соответствие модуль *trans1*. На подстранице, соответствующей этому переходу, присутствуют копии всех мест, которые связаны с модулем *trans1*, а именно: *State*, *queue*, *self*, *sender* и *c32&m22*. Моделирующая подсеть изображена на рис. 12.

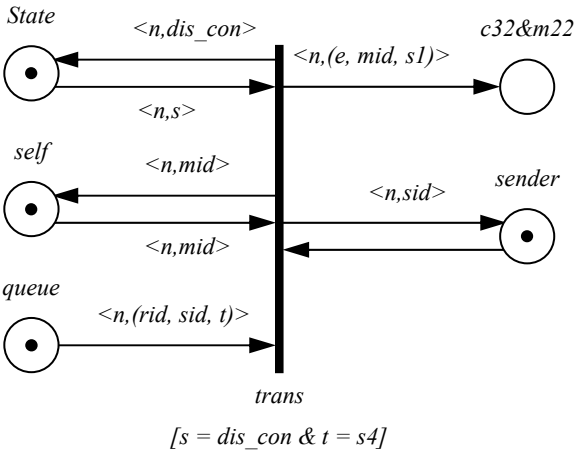

Рис. 12. Сеть, моделирующая первый переход процесса `Init`

Переход *trans* может сработать, если:

– место *State* содержит фишку, второе поле которой есть *dis_con*, что соответствует состоянию процесса `disconnected`;
– третье поле первого элемента в очереди в месте *queue* имеет значение *s4*.

Срабатывание перехода *trans* заключается в следующем:

– из места *State* из слоя с номером *n* забирается фишка со значением *(n, dis_con),* и она же в него возвращается;
– из места *queue* из слоя с номером *n* забирается из очереди первая фишка со значением *(rid, sid, s4)*;
– из места *self* забирается фишка со значением первого поля *n* и она же в него и возвращается;
– из места *sender* изымается фишка со значением первого поля *n,* и помещается фишка со значением *(n, sid)*;
– в место *c32&m22* добавляется фишка со значением *(e, mid, s1)*.

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. Каждое служебное место *connect* имеет пометку *слой*, второе поле фишки содержит значение *e*. 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 информация о том, что экземпляр процесса существует в системе, образуют спусковую функцию перехода *begin*.

На странице, представляющей SDL-переход, присутствуют места *self* и *sender*, которые являются входными и выходными для этого перехода, а также 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 слое, представляющему рассматриваемый экземпляр процесса, не может сработать никакой другой переход, кроме моделирующих этот SDL-переход

Однако в сети могут срабатывать N-переходы с наборами фишек, относящимся к другим слоям. 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 N-переходов, которые моделируют переходы других процессов.

В качестве примера рассмотрим второй переход процесса `Init` (рис. 10). Этот переход является сложным. Сеть для модуля *trans2* представлена на рис. 13. Фрагмент, состоящий из последовательности операторов

```
TASK counter := x;
OUTPUT s5;
OUTPUT s6;
```

в сети представляется одним переходом, назовем его *frg*. Подсеть для конструкции

```
SET(NOW+p, t);
```

на рисунке не показана, моделирование таймерных конструкций будет описано ниже.

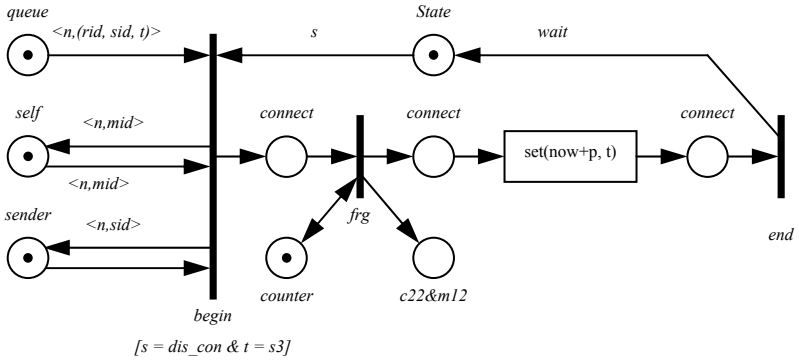

Рис. 13. Сеть, моделирующая второй переход процесса `Init`

Моделирование процедур и их вызовов происходит следующим образом. В сети для SDL-перехода вызову процедуры соответствует модуль. Места, моделирующие переменные, которые участвуют в вызове процедуры, являются входными и выходными местами для модуля, представляющего этот вызов. Процедуры и макрокоманды моделируются таким же способом, что и фрагмент тела перехода. Сеть, представляющая соответствующую процедуру/макрокоманду, размещается на странице, связанной с модулем, который представляет оператор вызова. При таком подходе отображение вызова процедуры, который содержится в теле другой процедуры, практически не отличается от отображения оператора вызова в SDL-переходе.

3.5. Моделирование доставки сигнала всем экземплярам процесса

Рассмотрим процесс, по описанию которого в системе создано несколько экземпляров. Доставка сигнала, отправленного этому процессу с помощью конструкции `VIA`, должна быть обеспечена всем экземплярам этого процесса. В сети, соответствующей такому процессу, строятся служебные места *count* и *count_copy*. Место *count* содержит фишку с целым значением, равным количеству созданных экземпляров моделируемого процесса, его

начальная разметка — фишка со значением *m*, где *m* — количество экземпляров процесса, созданных при инициализации системы. Значение этой фишки изменяется при создании нового экземпляра и при уничтожении существующего. Место *count_copy* хранит фишку целого значения — количество экземпляров, в порт которых требуется передать сигнал.

Напомним, что каждый переход *link* связывает место, моделирующее входной маршрут процесса, с местом *queue*. При его срабатывании сигнал из входного маршрута доставляется в очередь процесса, при этом копирования сигнала не происходит. Для доставки сигнала всем экземплярам процесса в сети необходимо реализовать специальный механизм, с помощью которого сигнал будет добавляться в место *queue* в конец очереди каждого слоя. Сеть, показанная на рис. 14, является подсетью сети, представляющей описание некоторого процесса, и моделирует пересылку сигнала *sig* из маршрута *m* всем экземплярам этого процесса.

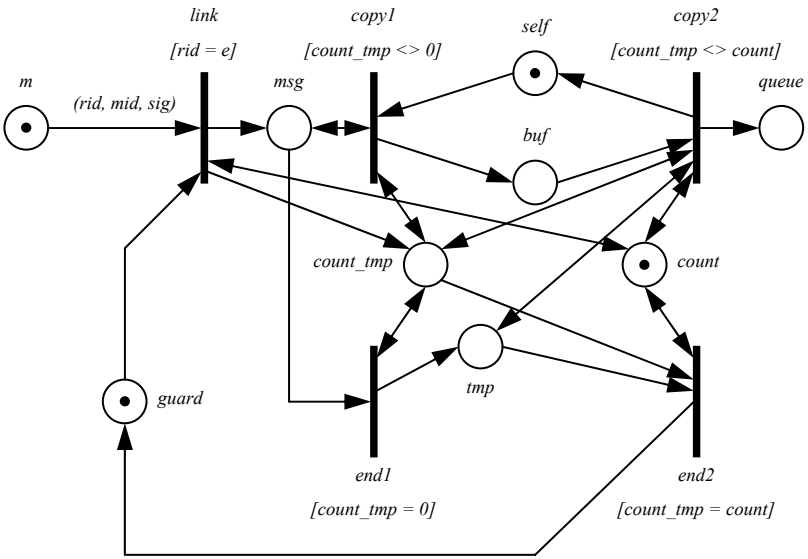

Рис. 14. Передача сигнала *sig* всем экземплярам одного процесса

Место *guard* гарантирует, что новое срабатывание перехода *link* не осуществится до тех пор, пока первое сообщение из маршрута не добавится в очередь каждого экземпляра процесса. Начальная разметка места *guard* —

одна фишка значения *e*. При срабатывании перехода *link* первое сообщение *(e, mid, sig)* забирается из очереди входного места-маршрута *m* и помещается в место *msg*, в место *count_copy* помещается фишка со значением, равным количеству функционирующих экземпляров процесса. Заметим, что при срабатывании переходов подсети, показанной на рис. 14, фишки из мест *count* и *count_copy* участвуют наравне с фишками любого слоя. Такие места будем называть *межслойными*.

Переход *copy1* сработает столько раз, сколько фишек имеется в месте *self*. При его срабатывании из места *msg* забирается фишка *(e, mid, sig)*, она же в него и возвращается; из места *self* забирается фишка с номером экземпляра процесса *n*; сообщение *(n, mid, sig)* помещается в место *buf* в слой с номером *n*; из места *count_copy* забирается фишка и возвращается в него со значением на единицу меньше предыдущего. Как только в месте *count_copy* будет находиться фишка со значением *0*, сработает переход *end1*. При его срабатывании изымается фишка из места *msg* и помещается в место *tmp*, после чего становится возможным переход *copy2*. При срабатывании перехода *copy,* изымается фишка *(n, mid, sig)* из места *buf* и добавляется в место *queue* в слой с номером *n*; в место *self* помещается номер экземпляра процесса, в очередь которого сообщение добавилось; в месте *count_copy* фишка заменяется на новую со значением на единицу больше предыдущего. Переход *copy2* сработает столько раз, сколько существует экземпляров процесса. После этого становится возможным переход *end2*. Его срабатывание изымает фишку из места *count_copy* и помещает фишку в место *guard*, что делает возможным новое срабатывание перехода *link*.

Если спусковая функция перехода *link* определяется идентификаторами каналов и маршрутов, то при его срабатывании из места-маршрута забирается первый элемент из очереди *(str, mid, sig)*, (где *str* — строка, состоящая из идентификаторов) и помещается в место *msg*, а далее выполняются переходы *copy1*, *end1*, *copy2* и *end2* так, как описано выше.

3.6. Моделирование оператора OUTPUT

Сложности при моделировании оператора `OUTPUT` возникают по той причине, что между экземплярами-братьями не существует каналов. Сигнал, отправленный брату, попадает в его порт. Таким образом, при моделировании посылки сигнала мы должны каждый раз проверять, отправляется сигнал экземпляру-брату или же какому-то другому процессу. Эта проверка осуществляется с помощью подсети, изображенной на рис. 16, а посылка сигнала — с помощью подсети, изображенной на рис. 15.

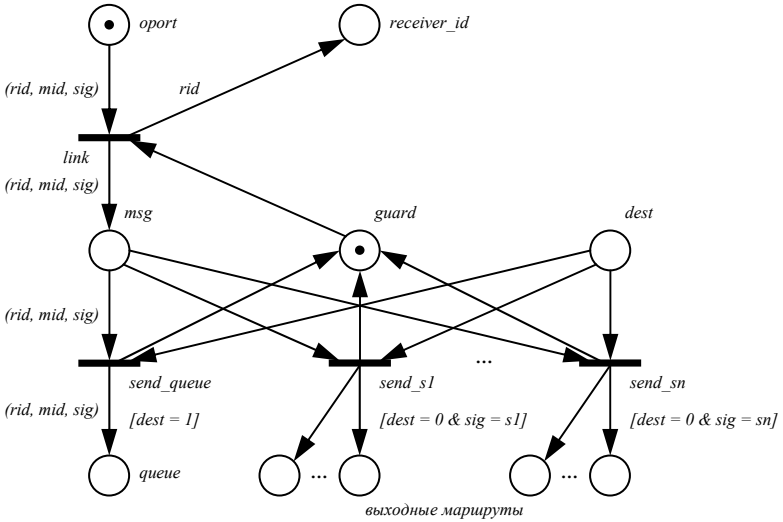

Рис. 15. Моделирование оператора `OUTPUT`

Места *oport*, *msg*, *guard*, *dest*, *tmp1* и *tmp2* являются межслойными местами в этих сетях. Моделирование оператора `OUTPUT` начинается с выполнения перехода, который помещает сообщение *(rid, mid, sig)* в место-очередь *oport* и который на рис. 15 не показан. Далее при срабатывании перехода *link* сообщение из места *oport* помещается в место *msg*; в место *receiver_id* помещается номер процесса-получателя; из места *guard* забирается фишка. После этого может функционировать подсеть, изображенная на рис. 16.

После выполнения переходов этой сети в место *dest* поместится фишка значения 1, если сообщение предназначается экземпляру-брату, иначе — 0. Далее может сработать один из переходов, моделирующих посылку сигнала. Если сообщение предназначается экземпляру-брату, то может выполниться переход *send_queue*, срабатывание которого перемещает фишку из места *msg* в место-очередь *queue* в слой с номером *rid*. Если значение фишки в месте *dest* равно 0, то может выполниться один из переходов *send*_s1, …, *send*_sn, где s1, …, sn — сигналы, которые могут передаваться данным процессом, а именно тот, у которого si = *sig,* где *i = 1*, …, *n*. Срабатывание этого перехода моделирует посылку сообщения в соответствую-

щий выходной маршрут.

Рассмотрим функционирование сети, показанной на рис. 16. Если сообщение предназначается конкретному адресату (т. е. *rid* не равен *e*), то может сработать переход *to_proper*. При его срабатывании изымается фишка из места *receiver_id* и добавляется в место *tmp1*; из места *count* забирается фишка со значением, равным количеству экземпляров данного процесса, она же в него и возвращается; в место *count_tmp* также помещается фишка с этим же значением. Далее может выполниться переход *check1*. При его срабатывании из места *tmp1* забирается фишка со значением *rid*, она же в него возвращается; из места *self* забирается фишка с номером экземпляра процесса n; номер экземпляра помещается в место *buf* в слой с номером *n*; из места *count_copy* забирается фишка и возвращается в него со значением на единицу меньше предыдущего. Переход *check1* может выполняться либо до тех пор пока не будут просмотрены все фишки в месте *self*, либо пока не найдется экземпляр, которому предназначается сообщение, в этом случае в сети значение n совпадет со значением *rid*.

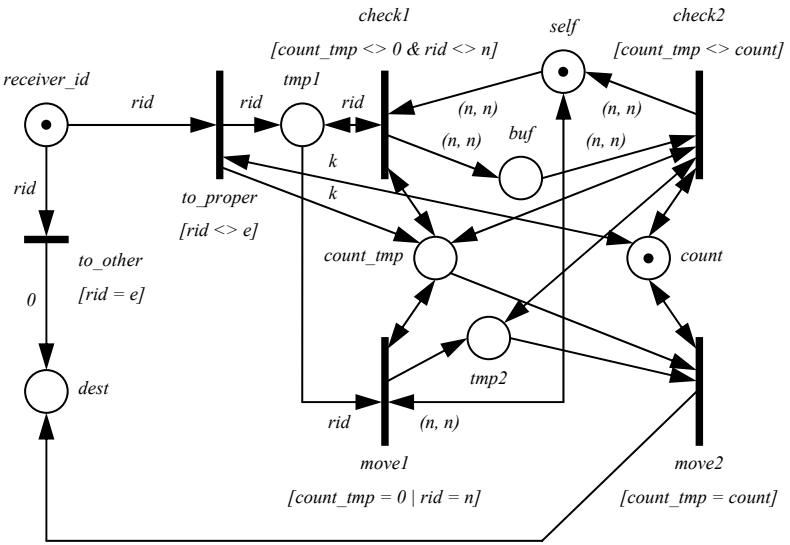

Рис. 16. Определение экземпляра-получателя

После этого может сработать переход *move1*. При его срабатывании фишка изымается из места *tmp1* и в место *tmp2* помещается фишка со значением 1 или 0, в зависимости от значения выражения (*rid* = *n*). После этого становится возможным переход *check2*. При его срабатывании фишка из места *buf* возвращается в место *self*; в месте *count_tmp* фишка заменяется на новую со значением на единицу больше предыдущего. Переход *check2* может срабатывать до тех пор, пока все фишки из места buf не будут возвращены в место *self*. После этого становится возможным переход *move2*. Его срабатывание изымает фишку из места *tmp2* и помещает фишку с тем же значением в место *dest*.

Если же сообщение *(rid, mid, sig)* предназначается не конкретному адресату, а любому процессу (в этом случае *rid* равно *e*), то становится возможным переход *to_other*, при срабатывании которого в место *dest* помещается фишка со значением 0. После этого выполнится один из переходов подсети (рис. 15) так, как было описано выше.

Таким образом, функционирование подсети на рис. 16 определяет, предназначается ли сообщение экземпляру-брату. Далее управление передается переходам подсети (рис. 15), моделирующим пересылку сообщения либо в очередь экземпляра-брата, либо в соответствующий выходной маршрут.

В семантике языка SDL не описано, что делается с сигналами, предназначенными еще не созданным экземплярам. Этот момент оставлен на волю реализации. Можно считать это ошибкой программиста или попытаться обойти такую ситуацию, например, следующим образом. В сети для каждого перехода, моделирующего оператор `OUTPUT`, создается дополнительный переход. Соединительное место *connect*, являющееся входным для перехода, представляющего оператор посылки сигнала, является входным и для дополнительного перехода. Аналогично можно сказать о выходном месте *connect*. Место *PId* также будет входным и выходным для него. Спусковая функция этого перехода определяется номером экземпляра, которому сигнал предназначается, и номером последнего созданного в системе экземпляра процесса. Последний хранится в месте *PId.* Срабатывание дополнительного перехода моделирует удаление из системы сигнала, который отправляется еще не созданному экземпляру процесса.

3.7. Моделирование порождения и уничтожения экземпляров процессов

Оператор `CREATE` в сети представляется тремя переходами: *generate*, *generate_nul* и *create*. Переходы *generate* и *generate_nul* располагаются в

сети, соответствующей родителю, для них выходными будут места *cr_id* (где *id* — имя порождаемого процесса), а место *PId* — входным и выходным. Переход *create* располагается в сети порождаемого процесса и для него служебные места *self*, *State*, *sender*, *parent*, места-переменные и места-параметры будут выходными. Переход *generate* может сработать в том случае, если число функционирующих экземпляров в сети не превысило максимально допустимого значения, иначе может сработать переход *generate_nul*. При срабатывании перехода *generate* из места *PId* забирается фишка со значением номера создаваемого экземпляра процесса, а в очередь в месте *cr_id* добавляется элемент, первое поле которого содержит личный идентификатор порождаемого экземпляра процесса, второе — личный идентификатор «экземпляра-родителя».

При срабатывании перехода *create* в сети создается новый слой с номером, равным личному идентификатору порождаемого экземпляра процесса. Это осуществляется путем добавления в каждое место, имеющее пометку *слой*, фишки, значение первого поля которой совпадает с ПИД этого экземпляра. Заметим, что срабатывание перехода *create* произойдет в том случае, если очередь в месте *cr_id* не пуста. Кроме того, если в сети имеется место *guard*, то его срабатывание зависит также от наличия фишки в нем. В месте *guard* фишка может отсутствовать при выполнении переходов сети (рис. 14), моделирующей доставку сигнала из входного маршрута всем экземплярам процесса.

При срабатывании переход *create*:

- помещает в каждое из выходных мест, кроме межслойных *count* и *guard*, по фишке, относящейся к слою, соответствующему личному идентификатору создаваемого экземпляра;
- изымает фишку из места *count* и возвращает в него фишку со значением на единицу больше предыдущего;
- изымает фишку со значением *e* из места *guard* и ее же возвращает.

Таким образом, место *guard* гарантирует, что во время функционирования сети, показанной на рис. 14, не произойдет создание нового экземпляра.

Фрагмент сети, моделирующей генерацию нового ПИД экземпляра, представлен на рис. 17. Предположим, что экземпляр процесса, осуществляющий создание процесса `A`, моделируется фишками, относящимися к слою с номером *n*.

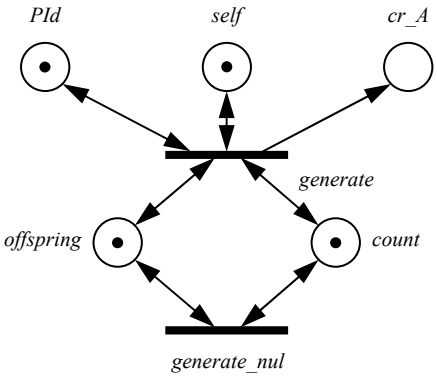

Рис. 17. Генерация нового ПИД

При срабатывании перехода *generate*:

- фишка с некоторым значением *p* забирается из места *PId* и возвращается в него со значением *(p + 1)*;
- из места *count*, находящегося на странице, моделирующей процесс `A`, забирается фишка и возвращается в него с тем же значением;
- из места *self* забирается и возвращается фишка со значением *(n, n)*;
- из места *offspring* забирается фишка и возвращается в него со значением *(n, p)*;
- к очереди в месте *cr_A* добавляется элемент *(p, n)*, где *p* — ПИД порождаемого экземпляра процесса `A`, *n* — ПИД экземпляра «процесса-родителя».

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

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 *n* 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 *n*. Входными к нему будут места *State*, *self*, *queue*, *sender*, *parent* и *offspring*, а также места-переменные, места-параметры и места, моделирующие таймеры процесса. Место *count* является входным и выходным для этого перехода. При его срабатывании фишка значения *k* замещается фишкой значения *(k - 1)*.

При моделировании сложного SDL-перехода места *State*, *self*, *queue*, *sender*, *parent*, *offspring* и места, моделирующие переменные, параметры и таймеры процесса будут входными для служебного перехода *end*. Место *count* будет входным и выходным для него. При срабатывании перехода *end* из каждого входного места заберется по одной фишке, относящейся к слою с номером *n*; в месте *count* фишка со значением *k* заменится на фишку со значением *(k – 1)*.

3.8. 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 сеть, представленную на рис. 18, которая моделирует сохранение всех сигналов в очереди некоторого экземпляра процесса, кроме сигнала `t`.

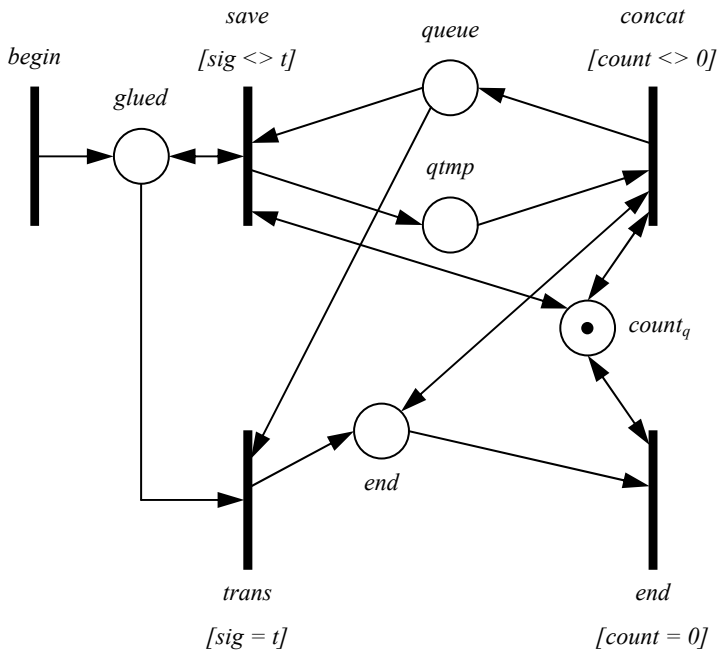

Рис. 18. Сеть, моделирующая конструкцию SAVE

Моделирование начинается с выполнения первого перехода *begin* и заканчивается выполнением перехода *end*. В сети созданы многослойные служебные места *glued* и *end*, которые могут содержать по одной бесцветной фишке в каждом слое, изначально они пусты. Также в сети содержится многослойное служебное место-очередь *qtmp*, которое предназначается для моделирования хранения сигналов, находящихся в очереди экземпляра процесса до сигнала `t`. Изначально это место пусто. Кроме того, сеть содержит межслойное служебное место *countq*, которое содержит целое число — длину очереди в месте *qtmp*, начальная разметка этого места — фишка со значением *0*. В подсети создано три перехода: *trans*, *save* и *concat*. Рассмотрим функционирование сети с фишками, принадлежащими какому-нибудь слою.

Как только в месте *glued* в этом слое появится фишка, в сети станет возможным либо переход *trans*, либо переход *save*. Здесь мы подразумеваем, что очередь сигналов процесса не пуста. Если в этом слое первая фишка в

месте *queue* содержит сигнал `t`, то может сработать переход *trans*, иначе — *save*. При срабатывании перехода *save* из места *queue* изымается первая фишка и добавляется в начало очереди в месте qtmp,, из места *glued* забирается фишка значения *e*, она же в него и возвращается, а в месте *countq* значение фишки увеличивается на единицу. Переход *save* может срабатывать до тех пор, пока на первом месте в очереди не окажется фишка с сигналом `t`. После нескольких срабатываний перехода *save* в месте *queue* останется хвост первоначальной очереди — элементы, моделирующие сигналы в очереди и находящиеся после сигнала `t`.

При срабатывании перехода *trans* из места *queue* забирается фишка с сигналом `t`, из места *glued* забирается бесцветная фишка и помещается в место *end*, в результате чего станет возможным переход *concat*. Он может сработать столько раз, сколько фишек имеется в месте qtmp,. При срабатывании перехода *concat* из места qtmp, забирается фишка и помещается в начало очереди в месте *queue*, в месте *count* значение фишки уменьшается на единицу. Переход *end* может выполниться в том случае, когда все элементы из очереди qtmp, переписаны в очередь *queue*, т. е. когда в месте *end* появится бесцветная фишка, а в месте *countq* — фишка со значением *0*.

Таким способом функционирование сети моделирует изъятие из очереди некоторого экземпляра процесса сигнала `t`. Заметим, что можно не использовать место *countq*, если задать переходу *concat* более высокий приоритет.

3.9. Моделирование конструкций разрешающего условия и непрерывного сигнала

Разрешающее условие — это булевское выражение, помещенное в угловые скобки после символа входа. Переход под воздействием некоторого входного сигнала совершается, только если значение булевского выражения равно *true*. В противном случае процесс переходит к обработке следующего сигнала в очереди, а предыдущий остается на своем месте.

Сеть, моделирующая SDL-переход с разрешающим условием, содержит в себе подсеть, представляющую конструкцию `SAVE`. В сети имеется дополнительный служебный переход, назовем его *serv*. Место *queue* является для него входным и выходным, а место *glued* — выходным. Если SDL-переход с разрешающим условием простой, то входной сигнал, состояние, из которого он возможен, и булевское выражение формируют его спусковую функцию, иначе — спусковую функцию служебного перехода *begin* в сети, моделирующей этот SDL-переход.

Спусковая функция перехода *serv* также определяется входным сигналом и булевским выражением. Он может сработать, если значение булевского выражения равно *false*. Его срабатывание помещает бесцветную фишку в место *glued*, в результате чего могут сработать переходы, моделирующие конструкцию `SAVE`. Таким образом, в случае ложности разрешающего условия входной сигнал останется в очереди на прежнем месте, а процесс перейдет к обработке следующего сигнала в очереди.

В конструкции непрерывного сигнала содержатся переходы, совершающиеся не под воздействием входных сигналов, а в силу истинности булевских выражений, входящих в эту конструкцию. На этапе моделирования процессов (пункт 3.3) было создано дополнительное служебное место, являющееся входным и выходным для каждого модуля, моделирующего SDL-переход в конструкции непрерывного сигнала. Начальная его разметка — одна бесцветная фишка. Это служебное место обеспечивает срабатывание только одного SDL-перехода из конструкции непрерывного сигнала. Если SDL-переход с булевским выражением простой, то это выражение преобразуется в спусковую функцию соответствующего N-перехода, а приоритет — в его пометку. При этом дополнительное служебное место будет для N-перехода входным и выходным. В противном случае булевское выражение преобразуется в спусковую функцию перехода *begin*, а служебное место будет для него входным и выходным для перехода *end* подсети, моделирующей данный SDL-переход. Значение пометки перехода *begin* совпадает с приоритетом соответствующего булевского выражения.

Если в конструкции непрерывного сигнала входной SDL-переход простой, то ему устанавливается приоритет выше установленного при моделировании SDL-перехода с булевским выражением, имеющим максимальный приоритет. В противном случае такой приоритет устанавливается служебному переходу *begin* подсети, моделирующей входной SDL-переход. Переходу *delete* устанавливается также максимальный приоритет. Все остальные N-переходы не помечаются, что соответствует наименьшему приоритету.

Таким образом, моделирование конструкции непрерывного сигнала отображается приоритетами в сети. Если очередь в месте *queue* не пуста, то в сети будут выполняться переходы, моделирующие входной SDL-переход, либо переход *delete*. Иначе — переходы, моделирующие тот SDL-переход в конструкции непрерывного сигнала, у которого истинно булевское выражение и максимальный приоритет.

3.10. Моделирование временных конструкций

Рассмотрим моделирование SDL-перехода, содержащего оператор установки таймера. Соответствующая сеть, представленная на рис. 19, имеет служебные переходы *begin* и *end* и подсети *Net1* и *Net2*, моделирующие фрагменты, соответственно находящиеся до и после оператора `SET` в теле SDL-перехода. В обведенном штриховой линией прямоугольнике находится подсеть, названная *Save* и моделирующая конструкцию `SAVE`. Чтобы не загромождать рисунок, некоторые места и переходы, а также дуги этой подсети не изображены, показаны лишь новые дуги, которые соединяют элементы этой подсети с другими элементами.

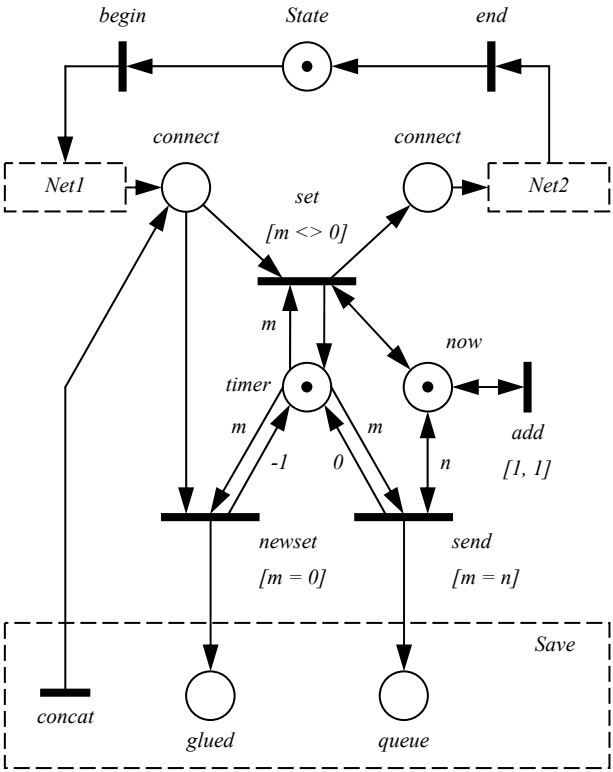

Рис. 19. Сетевое представление перехода с оператором `SET(m, t)`

На странице для SDL-перехода присутствует копия места *now*, которое было создано на этапе моделирования структуры системы (пункт 3.1). Напомним, что значение фишки в этом месте определяет текущий момент в системе, начальная разметка — одна фишка значения *0*. Переход *add* имеет интервал срабатывания *[1, 1]*. Как только он становится возможным, он ожидает одну единицу времени, а далее выполняется, при этом из места *now* фишка со значением текущего времени изымается и в него помещается фишка со значением, на единицу больше предыдущего.

Для отображения таймера `t` в сети создано многослойное место *timer*, множеством цветов второго поля служит множество целых чисел. Начальная разметка этого места — в каждом слое одна фишка значения *-1*, что соответствует тому, что таймеры у экземпляров процесса находятся в неактивном состоянии.

Переход *send* может сработать в том случае, если в некотором слое значение фишки в месте *timer* совпадает со значением фишки в месте *now*. Это говорит о том, что значение в таймере у соответствующего экземпляра процесса стало равно текущему времени в системе. Интервал срабатывания перехода *send* равен *[0, 0]*, т. е. переход обязан сработать сразу, как только станет возможным. Все остальные переходы в сети, кроме перехода *add* имеют интервал срабатывания *[0, ∞]*. Срабатывание перехода *send* заменит в данном слое фишку в месте *timer* на фишку со значением второго поля *0* и добавит фишку *(p, (p, p, t))* (где *p* — личный идентификатор моделируемого экземпляра процесса) в место *queue* в слой с номером *p*. Таким образом, фишка значения *(p, 0)* свидетельствует о том, что сигнал от таймера находится в очереди экземпляра процесса с ПИД равным *p*.

Потребление сигнала от таймера каким-либо простым SDL-переходом этого экземпляра процесса моделируется срабатыванием соответствующего N-перехода, которое в слое с номером *p* забирает из очереди фишку, соответствующую сигналу от таймера, и заменяет фишку в месте *timer* на фишку значения *(p, -1)*. После чего таймер этого экземпляра становится неактивным. Потребление сигнала от таймера каким-либо сложным SDL-переходом моделируется срабатыванием первого служебного перехода *begin* в подсети, моделирующей этот SDL-переход, которое также приводит к тому, что в месте *timer* окажется фишка значения *(p, -1)*.

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 предыдущей установкой таймера.

В моделирующей сети это осуществляется следующим образом. Если значение второго поля фишки в месте *timer* не равно *0* (т. е. сигнал от таймера еще не послан в очередь экземпляра процесса), может сработать переход *set*, в результате чего фишка в месте *timer* заменится на новую, соответствующую новому установленному времени. Иначе в сети может сработать переход *newset*. После его срабатывания возможно срабатывание переходов подсети *Save*, моделирующей изъятие из очереди экземпляра процесса сигнала от таймера. После их завершения в месте *timer* появится фишка значения *(p, -1)*, в результате чего становится возможным переход *set*.

Моделирование оператора `RESET` производится аналогичным образом.

4. РЕАЛИЗАЦИЯ АЛГОРИТМА МОДЕЛИРОВАНИЯ

Система SPV, представляющая собой интегрированный программный комплекс для проектирования, анализа и симуляции сетевых моделей распределенных систем, разработана в рамках проекта, нацеленного на валидацию коммуникационных протоколов. Основными компонентами комплекса являются транслятор с языка SDL, графический редактор и симулятор.Транслятор осуществляет автоматический перевод SDL-спецификации во внутреннее представление ИВТ-сети. Внутреннее представление ИВТ-сети имеет специальные пометки на дугах, переходах и местах. Алгоритм перевода SDL-спецификаций в сетевые модели реализован методом двухпроходной трансляции. На первом проходе строится внутреннее представление программы в виде атрибутированного дерева разбора, а на втором — генерируется сетевая модель. Синтаксический анализатор построен с помощью программного средства Bison.

Процесс построения сетей производится по уровням, соответствующим этапам в описании алгоритма трансляции. Все уровни сети, кроме нижнего уровня, используют модули системы SPV. Таким образом, вся сеть представляется деревом страниц системы SPV. Разработано специальное представление для отображения во внутренней структуре системы SPV перехода, места, дуги и их связей. Перед построением сети проводится предварительный анализ соответствующего синтаксического дерева спецификации.

При генерации ИВТ-сети в первую очередь создается корневой уровень иерархической сети, который состоит из модулей системы SPV, представляющих блоки SDL-спецификации. На этом же этапе создаются места, моделирующие каналы. Эти места и модули соединяются дугами в соответствии с соединением блоков с каналами. Если спецификация содержит вре-

менные конструкции, то создаются место *now* и переход *add*. При наличии в спецификации динамических конструкций создается служебное место *PId*. Следующие шаги генерации выполняются последовательно для каждого блока.

На втором шаге генерируется сеть, реализующая блок SDL-спецификации, состоящий из процессов. Эта сеть, в свою очередь, будет содержать:

- модули, соответствующие описаниям процессов;
- места, моделирующие маршруты, слитые с копиями мест-каналов в соответствии с алгоритмом;
- места, моделирующие экспортируемые и обозреваемые переменные;
- дуги, соединяющие места и модули.

Кроме того, сеть содержит копии мест *now* и *PId*.

На третьем шаге генерируется сеть, реализующая процессы, которая содержит модули, соответствующие переходам процесса; копии мест, моделирующих экспортируемые и обозреваемые переменные; места-переменные; служебные места; место текущего состояния процесса. Дуги, которые соединяют места, моделирующие переменные (в том числе экспортируемые и обозреваемые), не создаются, а достраиваются на следующем шаге. Если процесс создается динамически, то в сети для него строятся переходы *generate_id*, *generate_null*, *create* и место *cr_id*.

На следующем шаге осуществляется генерация сети для SDL-переходов, а далее транслируются отдельные операторы, а также создаются дуги для сети второго уровня.

При создании сетевой модели в системе SPV внутренние структуры содержат информацию, используемую при построении ИВТ-сети. После того как сеть создана, осуществляется размещение ее элементов на плоскости, т. е. каждому элементу приписываются координаты на соответствующих страницах системы SPV. При этом фрагменты сети, реализующие стандартные конструкции размещаются типовым образом.

4.1. 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 и экспортируемых), *par* параметров, *t* таймеров и имеет *m* входных и выходных маршрутов. Кроме того, пусть данный процесс содержит *n* операторов, среди которых *k1* операторов `DECISION`, *k2* операторов `SET`, *k3* операторов `RESET`, *k4* конструкций `JOIN`, *k5* конструкций

SET, *k3* операторов RESET, *k4* конструкций JOIN, *k5* конструкций SAVE и *C* вызовов процедур. Если в процессе используются конструкции SAVE, SET, RESET, то их моделирование требует дополнительных построений, добавляющих, самое большее, по 7 мест и по 6 переходов на каждую конструкцию.

Обозначим *k=k1+k2+k3+k4*, *att=var+14+2*m+par+t*. Тогда сеть, моделирующая описание процесса, будет иметь максимально *TN* переходов и *PN* мест, где

$$TN=(2*n+5+2*m+6*k)*(C+1) \text{ и } PN=(n+att+7*k)*(C+1).$$

Моделирующая сеть спецификации состоит из сетей, моделирующих описание процессов, мест-каналов, мест *now* и *PId*, а также дополнительных переходов, отображающих точку присоединения маршрутов к каналам, которых в сети не более чем *o*s*, где *o* — количество различных описаний процессов в спецификации, *s* — максимальное количество сигналов, передаваемых по каналу.

Таким образом, общая оценка размера сети представляет собой сумму оценок всех процессов, к которой добавляется *5*(o*(s+m)+1)+n* переходов; 6*(*m*p+1)* мест, *p* — максимальное количество различных экземпляров одного процесса.

ЗАКЛЮЧЕНИЕ

В настоящей работе предложены новая сетевая модель — многослойные ИВТ-сети и алгоритм перевода в эту модель спецификаций языка SDL. В ходе работы алгоритма все переменные отображаются в места. Причем в каждом слое в них содержится по одной фишке. Каналы и маршруты представляются местами, которые являются межслойными и содержат очереди сообщений. Состояние экземпляров процесса в SDL в каждом слое представляется единственным значением фишки в месте *State*. Служебные места *self*, *sender*, *glued*, *connect* также в каждом слое содержат по одной фишке. Порт каждого экземпляра процесса представляется очередью фишек в месте *queue*, причем в каждом слое может находиться не более одной очереди. Этот факт позволяет не производить 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 из разных слов, кото-

рых достаточно для двух или более одновременных срабатываний. Например, при моделировании таймера (рис. 19) в месте *timer* может находиться *n* различных фишек, моделирующих установленные в таймерах времена в *n* различных экземплярах одного процесса. Если окажется, что у каждой из фишек значение второго поля совпадает со значением фишки в межслойном месте *now*, то переход *send* должен сработать *n* раз в системе в один и тот же момент времени.

Обычно системы, моделирующие SDL, определяют фактическую семантику используемых конструкций и вносят определенные ограничения. Поэтому, можно считать, что предложенный алгоритм задает сетевую семантику базовых конструкций языка SDL.

Экспериментальная система SPV [22] используется для проектирования и симуляции ИВТ-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. Эксперименты, проведенные с протоколами, подтверждают эффективность полученных сетевых моделей, а оценки их размеров приемлемы для реальных 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. — Lect. Notes Comput. Sci. — 2003. — Vol. 2708. — P. 285–299.
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. — Lect. Notes Comput. Sci. — 2003. — Vol. 2679. — P. 450–462.
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. — Lect. Notes Comput. Sci. — 1998. — Vol. 1384. — P.102–117
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. — Lect. Notes Comput. Sci. — 1999. — Vol. 1708. — P. 642–661.
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-Verlag, 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, N 2. — P. 98–132.
16. **Nepomniaschy V.A. et al.** Modeling and verification of SDL specified distributed systems using high-level Petri nets // Proc. Workshop “Concurency, Specification and Programming” (CS&P’2004), Informatics-2004. — Berlin, 2004. — P. 100–111. — (Bericht / Humbold Univ.; N 170).
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.и др.** 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). — М.: 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; № 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) .
22. **Чурина Т.Г.** Трансляция SDL-спецификаций в раскрашенные сети Петри // IV сибирский конгресс по прикладной и индустриальной математике (ИНПРИМ-2000) Тез.докл. — Новосибирск: Ин-т математики СО РАН, 2000. — С. 128.

Т. Г. Чурина, В. С. Аргиров

МОДЕЛИРОВАНИЕ СПЕЦИФИКАЦИЙ ЯЗЫКА SDL С ПОМОЩЬЮ МОДИФИЦИРОВАННЫХ ИВТ-СЕТЕЙ

Препринт
124

Рукопись поступила в редакцию 25.09.05
Рецензент Е. Н. Боженкова
Редактор З. В. Скок

Подписано в печать 25.10.05
Формат бумаги 60 × 84 1/16 Объем 3.7 уч.-изд.л., 4 п.л.
Тираж 60 экз.

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383) 330-72-02