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

Т. Г. Чурина

МОДЕЛИРОВАНИЕ ДИНАМИЧЕСКИХ КОНСТРУКЦИЙ ЯЗЫКА SDL ПОСРЕДСТВОМ РАСКРАШЕННЫХ СЕТЕЙ ПЕТРИ

Препринт
71

Новосибирск 2000

Работа посвящена исследованию проблемы автоматического построения сетевых моделей динамических SDL-спецификаций распределенных систем. В качестве моделей выбраны раскрашенные сети Йенсена, обогащенные приоритетами. В работе представлен метод трансляции динамических конструкций SDL-систем в данную сетевую модель.

© Институт систем информатики им. А. П. Ершова СО РАН, 2000

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

T. G. Churina

COLOURED PETRI NETS APPROACH TO THE VALIDATION OF DYNAMIC SDL-SPECIFICATIONS

Preprint
71

Novosibirsk 2000

This work is concerned to research dealing with automated constructing net models of dynamic SDL specifications. Coloured Petri nets introduced by Jensen are used as the net models. Describes a method of translating the dynamic construction of SDL specification into the net models.

© A. P. Ershov Institute of Informatics Systems, 2000

ВВЕДЕНИЕ

В последние годы прогресс средств передачи и обработки информации привел к стремительному проникновению телекоммуникационных технологий во все сферы жизни, благодаря чему построение и верификация коммуникационных протоколов стала одной из активно развивающихся областей формальной верификации. Большие объемы и высокий уровень сложности разрабатывемых протоколов потребовали создания средств атоматической или полуавтоматической верификации распределенных асинхронных систем, в том числе и коммункационных протоколов.

Для Estelle-спецификаций предложен метод автоматического построения конечно-автоматных моделей посредством исчерпывающей симуляции [1], позволяющий верифицировать некоторые свойства коммуникационных протоколов. В работах [2, 3] представлены методы трансляции Estelle-спецификаций в сети Петри, причем в последней работе рассматривается широкое подмножество Estelle-спецификаций, включающее динамические конструкции, но без задержек и приоритетов, различные экземпляры модулей представлены фишками. Работа [4] посвящена исследованию проблем автоматического построения сетевых моделей Estelle-спецификаций распределенных систем и развития средств верификации этих моделей. В ней представлен метод трансляции Estelle-спецификаций в сетевую модель, описана его реализация, а также система *NetCalc*, предназначенная для редактирования и симуляции таких моделей.

Интересные методы трансляции SDL-спецификаций в обобщенные сети Петри описаны в работах [5, 6], в которых предложены методы построения графа достижимости. В работе [5] представлена техника анализа эффективности для SDL-сетей, однако, поскольку графы достижимости весьма громоздки, обычные способы их обработки неэффективны, для применения этих методов требуются дальнейшие исследования.

В книге Йенсена [9] (стр. 148) поставлены проблемы автоматического построения сетевых моделей SDL-спецификаций, развития средств их верификации, а также проведения экспериментов по обнаружению семантических ошибок распределенных систем с помощью этих средств.

Цель предложенной работы состоит в автоматическом переводе спецификаций языка SDL88 в раскрашенные сети Йенсена [7]. С другой стороны эту работу можно рассматривать как попытку создать сетевую

семантику для языка SDL. В работе [10] описан алгоритм трансляции SDL-спецификаций, не содержащих динамических конструкций, в раскрашенные сети Петри (РСП). В данной работе, являющейся продолжением работы [10], описано моделирование динамических конструкций языка SDL, а именно: запрос на создание другого процесса (оператора *create*) и механизм уничтожения процесса. В качестве модели выбраны раскрашенные сети Йенсена, обладающие временным механизмом. Эта модель была расширена приоритетами [11]. Способ моделирования основан на том, что многоуровневое описание системы в SDL имеет статический шаблон. Число экземпляров процесса может измениться в процессе функционирования системы, но позиция каждого экземпляра процесса в общей иерархии системы остается неизменной и соответствует положению процесса в структуре описаний системы. Статический шаблон будет моделироваться структурой сети, а моделирование экземпляров процесса будет происходить с помощью фишек, число которых может меняться.

Моделирование некоторых конструкций языка SDL, описанное в данной работе кратко, можно найти в [10]. Данная работа состоит из введения, 8 разделов и заключения. Первый раздел содержит описание функционирования процессов в языке SDL. Второй посвящен описанию раскрашенных сетей Йенсена. В третьем разделе изложены основные принципы трансляции SDL-систем, и описана трансляция верхних уровней для спецификаций, содержащих динамически создаваемые экземпляры процессов. В четвертом представлена трансляция процесса, пятый раздел содержит описание трансляции перехода. Шестой раздел посвящен моделированию запроса на порождение, а в седьмом описан способ "уничтожения"экземпляров процесса. В восьмом приводится моделирование средств управления временем.

1. ЯЗЫК SDL: ФУНКЦИОНИРОВАНИЕ ПРОЦЕССОВ

Международный консультативный комитет по телеграфии и телефонии разработал язык спецификаций и описаний SDL (Specification and Description Language), предназначенный для описания структуры и функционирования систем реального времени [12, 13]. Средствами этого языка обеспечивается многоуровневое описание системы. По мере "спуска"от уровня к уровню либо детализируются описания уже имеющихся в системе объектов, либо вводятся новые объекты.

Коротко напомним, что самый общий объект, описываемый на SDL,

называется *системой*, которая может состоять из одного или нескольких *блоков*, соединенных между собой и с окружающей средой *каналами*, по которым передаются *сигналы*. Каналы бывают одно- или двусторонними. На самом нижнем уровне иерархии определений блоки содержат *процессы*. Между собой и с рамкой блока процессы соединяются *маршрутами*.

Процесс в языке SDL — это обобщение понятия расширенного конечного автомата. Каждый процесс обладает конечным числом состояний, портом, в котором образуется очередь входных сигналов, и может обладать некоторым набором переменных. Процесс либо находится в одном из своих состояний, либо совершает переход. Переход из одного состояния в другое совершается под влиянием сигналов извне. При переходе процесс может выдавать выходные сигналы, манипулировать переменными и выполнять целый ряд действий.

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 инициации 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 личный идентификатор. Для этой цели каждый процесс обладает переменной $self$, которой при создании экземпляра процесса присваивается личный идентификатор этого экземпляра.

Каждый экземпляр процесса обладает еще тремя переменными: $parent$, $offspring$ и $sender$. Переменной $parent$ порожденного процесса присваивается значение переменной $self$ родительского процесса. Переменной $offspring$ родительского процесса присваивается значение переменной $self$ его последнего отпрыска. Переменной $sender$ присваивается личный идентификатор процесса-отправителя.

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

$$create\langle имя_процесса\rangle\langle фактические_параметры\rangle,$$

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.

В каждом блоке должен быть хотя бы один процесс, который возникает в момент инициации системы. Этот процесс может порождать другие процессы, но только в своем блоке. Если процесс хочет породить какой-то процесс в другом блоке, то он должен послать 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.

Процесс имеет одну стартовую вершину, за которой следует переход. Этот переход совершается не под влиянием входного сигнала, а в результате возникновения процесса. Дальнейшие переходы из состояния в состояние возможны только под воздействием входных сигналов.

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

1) явно указано, какой переход должен совершить процесс. Тогда процесс удаляет из очереди сигнал и начинает указанный переход;
2) 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*;
3) 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.

При переходе процесс может выполнить любой оператор языка Паскаль, а также такие действия, как принятие решения; выдачу выходных

сигналов; запрос на создание другого процесса; экспортно-импортную операцию; безусловный переход к другой последовательности действий; установку и сброс таймера; вызов процедуры.

За пределами этого рассмотрения языка SDL остались описания абстрактных типов данных, вышеуказанных действий, макросредств, присоединений маршрутов к каналам. Подробное описание этих конструкций приведено в работах [12, 13].

Ограничения на SDL. Трансляция будет проводиться для систем, в которых происходит динамическое создание и уничтожение экземпляров процессов, но которые не имеют указательных переменных.

В настоящий момент разработана процедура трансляции только для операторов отправления сигналов вида

$$Output\ \langle \text{имя сигнала} \rangle \langle \text{параметры} \rangle\ to\ \langle \text{личный идентификатор} \rangle,$$

и

$$Output\ \langle \text{имя сигнала} \rangle \langle \text{параметры} \rangle.$$

Первая конструкция означает, что сигнал, возможно с параметрами, посылается процессу с номером, указанным после служебного слова *to*. Вторая — всем процессам, связанным с процессом-отправителем маршрутами, которые могут передавать этот сигнал. Таким образом, в этой версии не использована конструкция передачи вида

$$Output\ \langle \text{имя сигнала} \rangle \langle \text{параметры} \rangle\ via\ \langle \text{имя маршрута} \rangle \ldots$$

$$\langle \text{имя маршрута} \rangle,$$

которая передает сигнал всем экземплярам тех процессов, с которыми процесс-отправитель связан поименованными в конструкции (после служебного слова *via*) маршрутами.

2. СЕТЕВАЯ МОДЕЛЬ

SDL-системы транслируются в иерархические раскрашенные сети, предложенные Йенсеном [7, 8] и обогащенные приоритетами [11]. Раскрашенная сеть состоит из трех частей: структуры сети, деклараций и пометки сети. Подробное описание можно найти в [14].

Структура сети представляет собой направленный граф с двумя типами вершин — *местами* и *переходами*, соединенными между собой дугами таким образом, что каждая дуга соединяет вершины различных

типов. Места имеют *разметку.* Ненулевая разметка места представляется наличием в месте одной или нескольких *фишек.* Кроме обычных мест, в сети могут присутствовать места–очереди.

Декларации состоят из описания типов (множеств цветов в терминологии Йенсена) и объявления переменных и, возможно, определения операций и функций.

Пометка сети приписывается месту, переходу либо дуге. Каждое место имеет три разных вида пометок: имя места, тип и инициализирующее выражение. Тип места определяет тип фишек, которые могут находиться в нем. При определении мест–очередей в раскрашенных сетях используется тип *list* (список). Инициализирующее выражение определяет начальную разметку места, т. е. какие фишки находятся в месте в начальный момент функционирования сети. К фишкам типа *list* применимы такие операции, как определение пустого списка, добавление некоторого элемента в список, помещение некоторого элемента в голову списка, конкатенации двух списков, взятие головы списка и другие.

Переход имеет четыре типа пометок: имя, спусковую функцию, приоритет и временную пометку — задержку. Спусковая функция перехода является логическим выражением, которое должно быть выполнено перед срабатыванием перехода. Приоритет перехода задается целым неотрицательным числом. Чем больше число, тем выше приоритет перехода. Отсутствие приоритета соответствует наименьшему приоритету. В сетевой модели имеется понятие глобальных часов, посредством которых в ней представляется текущее время. Временная пометка определяет момент времени, раньше которого фишки, помещаемые во все выходные места при срабатывании этого перехода, не могут использоваться. Таким образом, эти фишки дополнительно имеют еще один цвет, называемый *временным штампом.*

Дуга имеет два типа пометок: выражение и временная пометка. Выражение на дуге может содержать переменные, константы, функции и операции, определенные в декларациях. Тип выражения на дуге должен совпадать с типом места, с которым связана дуга. Временная пометка на дуге определяет момент времени, раньше которого фишка, помещаемая в место, в которое ведет данная дуга, не может быть использована.

Переходы в сети срабатывают, изменяя при этом разметку своих входных и выходных мест. Прежде чем переход сработает, все переменные, входящие в спусковую функцию перехода, и выражения на связанных с ним дугах, должны получить значения. Все вхождения одной

и той же переменной замещаются одним и тем же значением. Выбор конкретного значения для каждой переменной определяет *связывание*.

Переход имеет право сработать из некоторого состояния сети, если выполнены следующие условия:

– переход *возможен*, т. е. все его входные места не пусты, и спусковая функция перехода имеет истинное значение при связывании, определенном входными фишками; переход *реализуем по времени*, т. е. временные штампы фишек, участвующих при выбранном связывании, должны быть не больше, чем текущее время в модели. Текущее время в модели изменяется только после того, как сработают все реализуемые переходы;
– *приоритет* перехода должен быть не меньше, чем приоритет всех остальных возможных реализуемых по времени переходов.

Срабатывание перехода изымает по фишке из всех его входных мест и добавляет по фишке в каждое выходное место. Значения добавляемых фишек определяются выражениями на выходных дугах перехода. Временной штамп добавляемых фишек вычисляется как текущее время в модели плюс временная пометка на переходе и дуге, если они есть.

Иерархическая раскрашенная сеть — это композиция множества неиерархических сетей, называемых *страницами*. Страницы могут содержать переходы специального типа, которые представляются подсетью, располагающейся на отдельной странице (будем называть их подстраницами), и которая в свою очередь также может содержать переходы специального типа. Подстраница содержит копии всех мест, с которыми связан переход специального типа. Место-копия может быть входным местом для некоторого перехода на подстранице тогда и только тогда, когда его *прототип* является входным местом для специального перехода, представляющего подстраницу. Аналогично, только копия выходного места-прототипа может быть выходным местом некоторого перехода на подстранице. Поведение иерархической сети определяется поведенчески эквивалентной ей неиерархической сетью, получающейся при замещении всех переходов специального типа страницами, которые они представляют. При этом каждый такой переход 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.

3. ПОШАГОВАЯ ТРАНСЛЯЦИЯ SDL-СИСТЕМЫ В РСП

В работе [10] сетевая модель создавалась с помощью поэтапного уточнения. На первом этапе была построена сеть, которая располагалась на первой странице, соответствовала основной структуре системы и содержала по одному переходу для каждого блока. Каждый канал, связанный с блоком, представлялся в сети одним или двумя местами-очередями, в зависимости от того, являлся он одно- или двунаправленным. Фишки в полученных местах могли принимать значения из множества цветов, которое определялось сигналами, передаваемыми по соответствующим каналам. Изначально все места, порожденные по описаниям каналов, имели нулевую разметку. Соединение переходов и мест осуществлялось дугами, направление которых совпадало с направлением передачи сообщений.

На втором этапе осуществлялась трансляция блока, которая происходила таким же образом, что и трансляция всей системы в целом. Переходы, построенные на первом этапе, заменялись подсетями, которые соответствовали разбиению блока и располагались на связанной с этим переходом подстранице. При трансляции блока, состоящего из подблоков и внутренних каналов, каждому подблоку в подсети соответствовал один переход, каждому внутреннему каналу — одно или два места-очереди, в зависимости от того, одно- или двунаправленный канал. Трансляция блока, состоящего из процессов, происходила аналогично трансляции блока любого уровня иерархии. Каждому экземпляру процесса соответствовал один переход, каждому маршруту — одно или два места-очереди, в зависимости от того, каков маршрут — одно- или двунаправленный.

При трансляции процесса каждый переход, соответствующий экземпляру процесса, заменялся подсетью, которая представляла внутреннюю структуру данного процесса и которая располагалась на отдельной подстранице. На этом этапе каждому SDL-переходу в сети соответствовал один переход. На следующих этапах осуществлялась трансляция SDL-переходов.

Очереди, ассоциированные с портами процессов, представлялись местами-очередями, а сигналы, сохраняемые в этих очередях, — фишками. Кроме того, каждая такая фишка содержала информацию о том, какой экземпляр процесса ее отправил и какому экземпляру она предназначается.

Для представления предопределенных сортов в SDL декларации се-

ти изначально содержали соответствующие множества цветов:

$Colour\ Int = integer;$

$Colour\ Bool = boolean;$

$Colour\ Char = char.$

Корректность такого представления была обеспечена тем, что, по определению раскрашенных сетей, множества цветов эквивалентны типам в языках программирования. В каждой сети было определено множество цветов, состоящее из одного элемента, который не несет информации:

$$Colour\ E = with\ e.$$

Фишка со значением e называется *ординарной* или *бесцветной*. Бесцветные фишки использовались в сетях, моделирующих SDL-системы в служебных целях. Описания всех сигналов, которые передавались по каналам в системе, списков сигналов и описания переменных транслировались в декларации сети.

В следующих ниже пунктах описано, какие дополнительные построения и изменения (по сравнению с алгоритмом из [10]) требуется провести в сети, моделирующей SDL-систему с динамическими конструкциями.

3.1. Трансляция системы и блока

При трансляции SDL-системы, содержащей динамические конструкции, на первом этапе создается сеть аналогично тому, как это было сделано в работе [10], которая будет соответствовать основной структуре системы, располагаться на первой странице и содержать по одному переходу для каждого блока. Отличие в построении состоит в том, что в этой сети не будут использоваться дополнительные служебные места для организации дисциплины FIFO-очереди [16]. Каждому каналу в сети будет соответствовать место, тип которого есть $list$. Любая фишка в таком месте будет представляться списком, каждый элемент списка — записью, значение первого поля записи есть личный идентификатор экземпляра процесса-получателя, второго — личный идентификатор экземпляра процесса-отправителя, остальные поля будут соответствовать сигналу в очереди к экземпляру процесса-получателя. Поскольку в РСП очередь к экземпляру процесса представляется одной фишкой, то место, моделирующее порт процесса, будет входным и выходным для каждого перехода в сети, помещающего сигнал в очередь или изымающего сигнал из очереди.

Для моделирования личных идентификаторов динамически создаваемых экземпляров процессов на первом этапе должно быть создано служебное место, назовем его Pid, которое будет содержать одну фишку. Значение этой фишки есть целое число — личный идентификатор процесса, создаваемого динамически во время функционирования сети. Начальная разметка этого места — фишка значения $n+1$, где n — количество экземпляров процессов, созданных в момент инициации системы. Это место будет входным и выходным местом для всех переходов сети (N-переходов), которые моделируют блоки, содержащие как процессы, выполняющие запрос на создание другого процесса, так и порождаемые процессы.

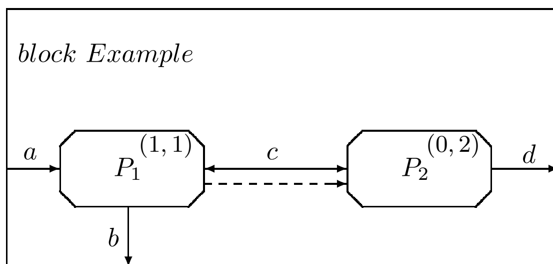

Рис. 1. Описание блока *Example*

На следующих этапах осуществим построение сетей для блоков также в соответствии с алгоритмом для SDL-спецификаций без динамических конструкций. Переходы, построенные на предыдущих этапах, заменятся подсетями, которые будут соответствовать разбиению блоков и располагаться на отдельных подстраницах. Служебное место Pid будет входным и выходным местом для всех N-переходов, моделирующих подблоки, которые содержат процессы, выполняющие запрос на создание другого процесса, и порождаемые процессы. На этапе трансляции блока, состоящего из процессов, место Pid также будет входным и выходным местом для всех N-переходов, моделирующих процессы, выполняющие запрос на создание другого процесса, и порождаемые процессы.

Кроме того, на этом этапе трансляции для каждого N-перехода, который моделирует процесс, содержащий оператор $create$, будет создано служебное место, назовем его cr, являющееся выходным местом для этого N-перехода и входным для N-перехода, моделирующего порождаемый процесс, указанный в операторе $create$. Фишки в месте cr могут

принимать значения из множества цветов $Create_Pid$, который определяется следующим образом:

$$Create_Pid = product\, Int * Int.$$

Значение первого поля фишки в месте *cr* есть личный идентификатор создаваемого оператором *create* экземпляра процесса, значение второго поля — личный идентификатор “экземпляра-родителя”.

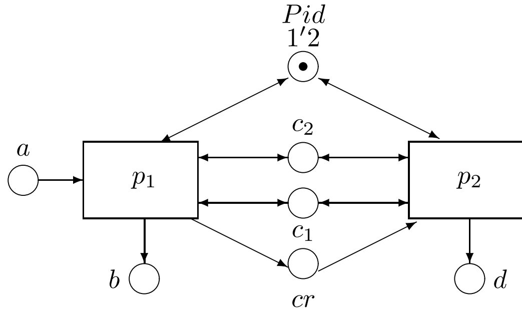

Рис. 2. Сеть для блока *Example*

Для приведенного на рис. 1 описания блока *Example* соответствующая сеть показана на рис. 2, служебное место *cr* является выходным для перехода p_1 и входным для p_2, моделирующими процессы P_1 и P_2 соответственно. Двунаправленный канал c, соединяющий блоки P_1 и P_2, в сети представляется местами c_1 и c_2. Место *Pid* будет входным и выходным местом для переходов p_1 и p_2.

4. ТРАНСЛЯЦИЯ ПРОЦЕССОВ

Коротко напомним алгоритм из работы [10], который применялся на этапе трансляции процессов. По этому алгоритму для каждого экземпляра процесса были созданы следующие служебные места: *State*, *queue*, *self*, *sender* и *stop*. Место *queue* соответствовало порту экземпляра процесса и содержало очередь фишек, соответствующих сигналам, поступающим по всем маршрутам к данному экземпляру. Служебное место *State* содержало фишку, цвет которой определялся состояниями экземпляра процесса, и обеспечивало следующее: пока срабатывали переходы сети, моделирующие один SDL-переход, были невозможны переходы сети, моделирующие другие SDL-переходы этого экземпляра про-

цесса. Начальная разметка этого места определялась одной бесцветной фишкой. Служебное место $self$ соответствовало переменной $self$ этого экземпляра, начальная разметка — номеру, который присваивался процессу после создания всей сети. Служебное место $sender$ соответствовало переменной $sender$. Начальная разметка этого места определялась одной фишкой значения 0. На этом этапе для каждого SDL-перехода экземпляра процесса в сети был создан один переход.

Для моделирования изъятия из очереди экземпляра процесса первого сигнала либо предназначенного другому экземпляру, либо невоспринимаемого этим экземпляром в том состоянии, в котором он находится, в сети был создан служебный переход del. Места $State$ и $self$ являлись входными и выходными местами для этого перехода, а место $queue$ — только входным.

Для моделирования "уничтожения" экземпляра процесса в соответствующей сети были созданы служебные место *stop* и переход *stp*. Место *stop* было входным и выходным для переходов *stp* и del, а место $queue$ — только входным. Кроме того, место *stop* было входным и выходным местом для всех переходов, моделирующих SDL-переходы на этом этапе. Начальная разметка места *stop* определялась одной фишкой значения 1. Спусковая функция перехода *stp* определялась сигналом, находящимся первым в очереди экземпляра процесса и являющимся таким, под воздействием которого процесс переходит в состояние *stop*. При срабатывании этого перехода в место *stop* помещалась фишка значения 0.

Переходу $START$ в сети соответствовал служебный переход $start$.

4.1. Трансляция экземпляров процесса

Как было отмечено ранее, многоуровневое описание системы в SDL имеет статический шаблон. Число экземпляров процесса может измениться в процессе функционирования системы, но позиция каждого экземпляра процесса в общей иерархии системы остается неизменной.

При построении моделирующей сети по каждому описанию процесса будет построена сеть аналогично тому, как это описано выше с той лишь разницей, что фишки во всех местах, кроме мест, о которых будет сказано особо, будут представляться записями, первое поле каждой записи будет иметь значение, соответствующее личному идентификатору экземпляра процесса. Различные экземпляры одного процесса будут моделироваться одной сетью, построенной по описанию этого процесса, но различными наборами фишек в местах этой сети. Каждая перемен-

ная в сети будет представляться местом, в котором первое поле любой фишки будет иметь цвет, соответствующий личному идентификатору экземпляра процесса, остальные поля — сорту переменной. Спусковые функции переходов сети дополнительно будут иметь условие, что каждая из фишек, определяющих связывание, относится к одному и тому же экземпляру процесса.

Все места в строящейся сети, моделирующей экземпляры процесса, создающиеся во время функционирования системы, изначально не будут иметь никаких фишек. Таким образом, будет заготовлена только "сеть-шаблон", в которой фишки появятся только после того, как в сети, соответствующей "процессу-родителю", сработает переход, моделирующий оператор $create$. Какие для этого нужны дополнительные построения в сети на этом этапе, будет подробно описано в п. 6. А сейчас отметим только, что в сети, соответствующей описанию процесса, экземпляры которого создаются динамически, должен быть построен служебный переход, назовем его $create$, срабатывание которого добавит по одной фишке во все служебные места и места, соответствующие переменным процесса. Первое поле каждой такой фишки будет содержать личный идентификатор создаваемого экземпляра процесса. Место cr будет для этого перехода входным.

Фишки в местах $sender$ и $stop$ будут принимать значения из множества цветов $pair$, где $Colour\ pair = product\ Int\ *\ Int$. Каждая фишка в месте $State$ будут представляться записью, значение первого поля фишки есть личный идентификатор экземпляра процесса, а значение второго поля будет определяться состояниями этого экземпляра процесса.

В сети, соответствующей процессу, экземпляры которого создаются в момент инициации системы, изначально в служебных местах будет столько фишек, сколько экземпляров процесса создано. Каждая фишка в месте $sender$ будет иметь значение $(n, 0)$, в месте $stop$ — значение $(n, 1)$, в месте $State$ — значение (n, e), в месте $queue$ — значение nil, где n — личный идентификатор экземпляра. Фишки в местах, соответствующих переменным, есть записи, первое поле которых будет иметь значение, соответствующее личному идентификатору экземпляра процесса, остальные — сорту переменной.

Место Pid будет входным и выходным местом для служебного перехода del. В спусковой функции этого перехода будет добавлено условие, контролирующее изъятие из каждого входного места по фишке, моде-

лирующей один экземпляр процесса. Также место Pid будет входным и выходным местом для всех N-переходов, моделирующих SDL-переходы, в которых имеется оператор $create$.

Если переход процесса выполняет запрос на создание экземпляра процесса, то на этапе трансляции процесса будут также созданы служебные места $parent$ и $offspring$, моделирующие соответствующие переменные в языке SDL. Фишки в них будут принимать значения из множества цветов $pair$. Изначально эти места пусты.

В сети, соответствующей процессу, экземпляры которого создаются во время функционирования системы, будет создан служебный переход $cancel_b$. Места Pid и $queue$ будут для него входными и выходными. Этот переход будет моделировать удаление из очереди каждого экземпляра процесса сигнала, находящегося в очереди первым и предназначенного еще несозданным экземплярам. Его спусковая функция будет определяться номером последнего созданного в системе экземпляра процесса.

В строящейся сети на этом этапе будет создан служебный переход $cancel_e$, для которого места $queue$ и $stop$ будут входными и выходными. Этот переход будет моделировать удаление сигнала, пришедшего к “уничтоженному” экземпляру процесса. Его спусковая функция будет определяться личным идентификатором “уничтоженного” экземпляра процесса. В сети при срабатывании этого перехода из места $queue$ заберется первый элемент списка, соответствующего очереди к “уничтоженному” экземпляру процесса. Если этот список не пуст, снова может сработать переход $cancel_e$.

Предположим, что процесс P_2 (см. рис. 1), экземпляры которого будут создаваться динамически, имеет два перехода, не содержащие оператора $create$, установку и сброс таймера. Пусть во втором переходе процесса осуществляется только прием некоторого сигнала из очереди процесса и посылка сигнала в выходной маршрут. В строящейся сети на этом этапе каждому SDL-переходу будет соответствовать один переход. На рис. 3 эти переходы названы $trans_1$ и $trans_2$.

Применение правил трансляции приводит к сети, изображенной на рис. 3. Поскольку экземпляры этого процесса создаются в процессе функционирования системы, то в сети на этом этапе будет создан переход $create$. Служебное место cr является для него входным, а все остальные служебные места — выходными. Чтобы не загромождать рисунок, используется двунаправленная стрелка для обозначения того, что место для перехода является как входным, так и выходным. Сеть содержит

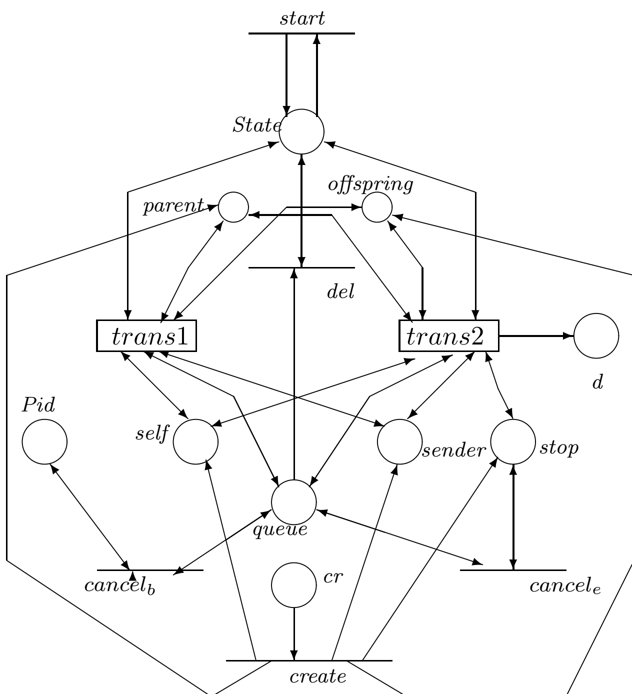

Рис. 3. Сеть для процесса P_2

служебные места *queue*, *State*, *self*, *sender*, *parent* и *offspring*.

Рассмотрим соединение мест в этой сети с переходом *trans*2. Все служебные места для перехода *trans*2 являются входными и выходными. Поскольку второй переход процесса посылает сигнал в выходной канал, то соответствующее этому каналу место *d* является выходным местом для перехода *trans*2.

Переход *start* соответствует начальному переходу процесса *START*. Переход *del* моделирует изъятие из порта процесса такого сигнала, который стоит первым в очереди к некоторому экземпляру процесса, но не воспринимается этим экземпляром процесса в том состоянии, в котором он находится. Для перехода $cancel_b$ места *Pid* и *queue* будут входными и выходными. Для перехода $cancel_e$ места *stop* и *queue* будут входными и выходными.

5. ТРАНСЛЯЦИЯ ПЕРЕХОДОВ

На этапе трансляции процесса для каждого SDL-перехода процесса в сети был создан один переход. На этом этапе каждый переход в сети, моделирующий SDL-переход, будет заменен подсетью, которая будет располагаться на связанной с этим переходом подстранице.

5.1. Трансляция простого перехода процесса

Как было отмечено в работе [10], выполнение некоторых SDL-переходов может моделироваться одним переходом сети, что происходит в том случае, если переход не содержит таких действий, как принятие решений, переход на метку, установка или сброс таймера, сохранение сигнала, разрешающее условие и вызов 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 переходы были называны *простыми*, и для них на этапе трансляции перехода были определены спусковые функции и выражения на входных и выходных дугах. Спусковая функция перехода сети, моделирующего простой переход процесса, определялась состоянием, из которого возможен этот SDL-переход, входным сигналом, указываемым в операторе *input*, личным идентификатором процесса. Выражения на дугах определялись состоянием, следующим после ключевого слова *nextstate*, и операторами, составляющими переход процесса.

В этом алгоритме спусковая функция каждого перехода сети, моделирующего простой SDL-переход, дополнительно будет содержать условие, которое при связывании, определенном входными фишками, контролирует, что эти фишки моделируют один экземпляр процесса.

Служебное место *stop* будет входным и выходным местом для всех переходов в сети, моделирующих простые SDL-переходы процесса. Допустим, что некоторый экземпляр процесса имеет личный идентификатор n. Тогда наличие фишки значения $(n, 1)$ в месте *stop* в сети, соответствующей описанию этого процесса, означает, что экземпляр процесса с номером n создан и функционирует. Наличие же фишки $(n, 0)$ в этом месте свидетельствует о том, что этот экземпляр "уничтожен". Спусковая функция каждого перехода в сети, моделирующего простой SDL-переход процесса, будет зависеть от значения второго поля фишки в месте *stop*.

Заметим, что состояние экземпляра процесса в SDL характеризуется собственно его состоянием, содержимым очередей, ассоциированных с маршрутами, и значением всех переменных. При трансляции состояние процесса отображается разметкой моделирующей сети. Построение происходит таким образом, чтобы срабатывание моделирующего перехода было возможно при некоторой разметке тогда и только тогда, когда в соответствующем состоянии экземпляра процесса может выполниться рассматриваемый SDL-переход. Определение РСП гарантирует, что выполнение простого перехода и срабатывание соответствующего ему перехода сети приводят к эквивалентным изменениям в состоянии экземпляра процесса и разметке сети соответственно.

5.2. Трансляция сложного перехода

SDL-переходы, содержащие конструкции *decision*, *join*, *set*, *reset*, *save*, вызовы процедур, разрешающее условие, а также операторы, использующие измененные в процессе выполнения перехода переменные, были названы *сложными*. Для них процесс трансляции продолжался. На последующих этапах трансляции сложный переход разбивался на фрагменты. Фрагмент сложного перехода мог быть вызовом процедуры, операторами *set*, *reset*, *save*, *decision* или простым оператором.

Каждому фрагменту в сети сопоставлялся переход. Данные переходы последовательно соединялись в том же самом порядке, что и соответствующие фрагменты, с помощью дополнительных служебных мест. Кроме того, были созданы два служебных перехода, один из которых становился первым, а второй — последним в цепочке переходов, моделирующих сложный SDL-переход. Первый служебный переход был назван *begin*, а второй — *end*. Они присоединялись к цепочке переходов, моделирующих фрагмент, с помощью служебных мест. Каждое служебное место могло содержать бесцветную фишку.

После того как переход был разбит на фрагменты, трансляция каждого из них осуществлялась отдельно. Процесс трансляции завершался, когда каждый из полученных фрагментов представлялся одним сетевым переходом. Таким образом, выполнение сложного SDL-перехода в сети моделировалось последовательным срабатыванием всех моделирующих его переходов, начиная с перехода *begin* и заканчивая переходом *end*.

В данном алгоритме к сложным переходам отнесутся все SDL-переходы, которые к ним относились и ранее, а также переходы, содержащие оператор *create*. Процесс выделения фрагментов на этом этапе будет вестись аналогично тому, как это осуществлялось в алгоритме из [10], но теперь фрагментом сложного перехода может быть оператор *create*. А каждое соединительное служебное место может содержать фишку значения (n, e), где n — личный идентификатор экземпляра процесса. Трансляция фрагмента с запросом на порождение другого процесса будет проведена так, как описано в разделе 6.1.

Место *State* по-прежнему будет являться входным для первого служебного перехода и выходным для второго служебного перехода. Состояние, указанное после служебного слова *State*, преобразуется в спусковую функцию первого служебного перехода. Кроме того, в спусковой функции этого перехода будет осуществляться проверка: принадлежат ли одному экземпляру процесса выбранные из входных мест фишки. Фишка значения (n, s) (где n — личный идентификатор, s — состояние процесса) из места *State* забирается, как только срабатывает первый служебный переход, и возвращается в него после срабатывания второго служебного перехода. Таким образом, до тех пор пока фишка не будет возвращена в место *State*, переходы сети, моделирирующие другие SDL-переходы, не могут сработать при связывании, в котором будут участвовать фишки, принадлежащие экземпляру процесса с личным идентификатором n. Это говорит о том, что для этого экземпляра процесса в моделирующей сети не может сработать никакой переход, кроме моделирующих рассматриваемый SDL-переход.

Наличие в сети места *State* превращает срабатывание нескольких переходов, соответствующих одному и тому же SDL-переходу некоторого экземпляра процесса, в непрерывное действие. Однако в процессе выполнения этих переходов сети ничто не мешает выполнению этих же переходов сети при связывании, в котором будут участвовать фишки, принадлежащие другому экземпляру этого же процесса, или таких переходов сети, которые моделируют переходы других процессов. Но это не влияет на правильность выполнения всей системы, так как выполнение разных экземпляров процесса независимо.

6. ПОРОЖДЕНИЕ ПРОЦЕССОВ

Рассмотрим моделирование SDL-перехода, содержащего запрос на порождение некоторого экземпляра процесса. В сети оператор $create$ будет представлен двумя переходами. Один переход, назовем его $generate$, будет принадлежать подсети, соответствующей этому SDL-переходу, а другой — $create$ будет принадлежать подсети, соответствующей порождаемому процессу.

Служебные места Pid и $self$ будут входными и выходными местами для перехода $generate$, а места $offspring$ и cr — только выходными. При срабатывании перехода $generate$ из места Pid заберется фишка, значение которой есть номер создаваемого экземпляра процесса, и вернется в него со значением на единицу больше этого номера; из места $self$ заберется фишка, значение которой есть номер “процесса-родителя”, и вернется в него с тем же значением; в выходные места $offspring$ и cr поместится фишка (p, l). Таким образом в место cr помещается фишка, значение первого поля которой есть Pid создаваемого экземпляра процесса, а второго — Pid порождающего экземпляра.

Служебное место cr будет входным местом для перехода $create$ в подсети, моделирующей порождаемый процесс, а все служебные места (кроме соединительных) и места, соответствующие переменным, — выходными. Если процесс содержит таймеры, то моделирующие их места также будут являться выходными для этого перехода (см. 8). При срабатывании перехода $create$ из места cr заберется фишка (p, l) и

– в место $parent$ поместится фишка значения (p, l), что соответствует тому, что у экземпляра процесса с личным идентификатором p “процесс-родитель” имеет личный идентификатор l;

– в место $sender$ добавится фишка $(p, 0)$;

– в место $self$ добавится фишка p;

– в место $State$ добавится фишка (p, e);

– в место $Stop$ добавится фишка $(p, 1)$;

– в место $queue$ добавится фишка nil;

– в каждое место, моделирующее переменную процесса, добавится фишка, значение первого поля которой есть p, остальные поля будут соответствовать начальному значению переменной, если оно определено, или будут равны нулю, если не определено;

– в каждое место, моделирующее таймер процесса, добавится фишка значения $(p, -1)$.

6.1. Пример

Ниже описан переход процесса P_1, содержащий оператор *create* P_2. (Процесс P_2 здесь не описан).

```
PROCESS P1(1,1);
............
STATE s1;
INPUT Sig;
CREATE P2;
NEXTSTATE s2;
..........
ENDPROCESS P1;
```

Этот SDL-переход моделируется переходом *generate* (см. рис.4) в подсети, соответствующей процессу P_1 и переходом *create* (см. рис.5) в подсети, соответствующей процессу P_2.

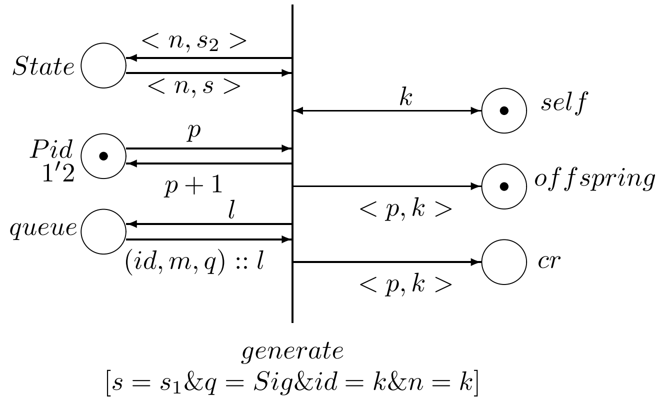

Рис. 4. Переход *genarate* в подсети для процесса P_1

Кроме описанных выше служебных мест, место *queue*, моделирующее порт процесса, и место *State* будут входными и выходными местами для перехода *generate*. Спусковая функция перехода *generate* делает возможным его срабатывание, если:

– место *State* содержит фишку значения (n, s_1), что соответствует состоянию s_1 экземпляра процесса с личным идентификатором n;

– место *queue* содержит фишку, третье поле первого элемента которой имеет значение Sig;

– первое поле каждой из фишек, определяющих связывание, замещается одним и тем же значением.

Срабатывание перехода *generate* заключается в следующем:

– из места *State* забирается фишка со значением (n, s_1) и возвращается в него со значением (n, s_2);

– из места *queue* забирается первый элемент (id, m, q) списка, соответствующего очереди к экземпляру процесса с номером id;

– из места *Pid* забирается фишка со значением p и возвращается в него со значением $p + 1$;

– из места *self* забирается фишка со значением k и в него возвращается;

– в место *cr* помещается фишка значения (p, k), где p — Pid порождаемого экземпляра процесса, k — Pid экземпляра процесса P_1;

– в место *offspring* помещается фишка со значением (p, k).

После срабатывания этого перехода в месте *cr* появится фишка, первое поле которой будет содержать *Pid* порождаемого экземпляра процесса, второе — *Pid* “экземпляра-родителя”.

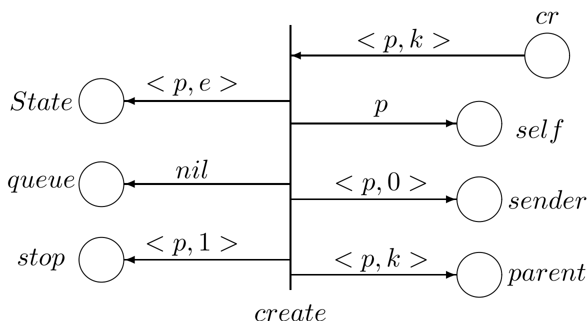

Рис. 5. Переход *create*

После этого в сети, моделирующей процесс P_2, становится возможным служебный переход *create* (см. рис.5), срабатывание которого заключается в следующем:

– из места *cr* забирается фишка (p, k);

– в место $State$ добавляется фишка (p, e);
– в место $self$ добавляется фишка p;
– в место $sender$ добавляется фишка $(p, 0)$;
– в место $parent$ добавляется фишка (p, k);
– в место $stop$ добавляется фишка $(p, 1)$;
– в место $queue$ добавляется фишка nil.

7. МОДЕЛИРОВАНИЕ “УНИЧТОЖЕНИЯ” ПРОЦЕССА

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*. На этапе трансляции процесса в сети было создано служебное место *stop*. Второе поле любой фишки в этом месте может иметь значение либо 1, либо 0. Фишка значения $(n, 1)$ гарантирует, что экземпляр процесса с личным идентификатором n в системе функционирует.

Рассмотрим переход процесса в языке SDL, при выполнении которого некоторый экземпляр входит в состояние *stop*. Если этот SDL-переход моделируется одним переходом в сети, то его спусковая функция будет определяться сигналом, указанным после служебного слова $INPUT$ в этом переходе, состоянием, из которого возможен переход, информацией о том, существует ли данный экземпляр процесса в системе, и условием, осуществляющим проверку того, что все выбранные из входных мест фишки моделируют один экземпляр процесса. Все служебные места и места, моделирующие переменные, будут входными для этого перехода. При срабатывании моделирующего перехода в место *stop* поместится фишка, значение первого поля которой есть личный идентификатор выбранного экземпляра процесса, значение второго поля равно 0, а из каждого входного места заберется по одной фишке, принадлежащей этому экземпляру процесса.

При моделировании сложного SDL-перехода спусковая функция первого служебного перехода *begin* будет зависеть от значения фишек в месте *stop*. Служебное место *stop* будет входным и выходным для второго служебного перехода *end*. Кроме того, служебные места $State$, $self$, $sender$,$parent$, $offspring$ и места, моделирующие переменные процесса, будут для него только входными. При срабатывании перехода *end* из каждого входного места заберется по одной фишке, принадлежащей

одному экземпляру процесса, а в место *stop* добавится фишка, значение первого поля которой есть личный идентификатор этого экземпляра, а значение второго поля равно 0.

Ниже описан переход процесса, воспринимающий сигнал *kill*, под влиянием которого процесс переходит в состояние *stop*.

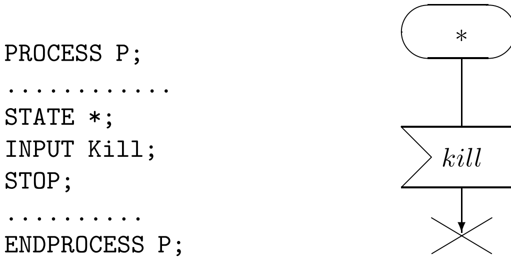

Рис. 6. Переход процесса в состояние *stop*

Цвет *kill* в декларации сети будет определен как
$Colour\ kill = withKill$.

Описанный SDL-переход в сети моделируется одним переходом, назовем его *remove* (см. рис.7). По построению сети место *stop* будет входным и выходным для перехода *remove*. Служебные места *queue*, *State*, *self*, *sender* будут для этого перехода только входными.

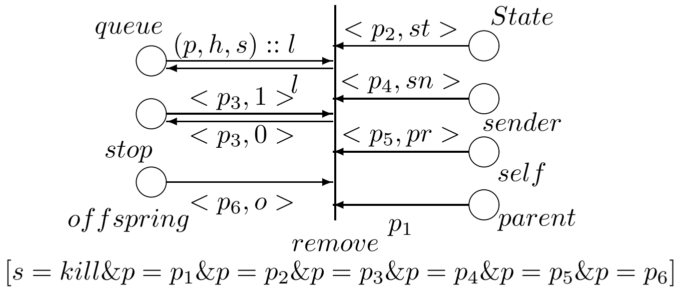

Рис. 7. Сетевое представление “уничтожения” экземпляра процесса

Рассмотрим некоторую фишку в месте *queue*, моделирующем очередь входных сигналов к экземплярам одного процесса, содержащего

описанный SDL-переход. Эта фишка представляет список, первое поле каждого элемента которого содержит личный идентификатор принимающего сигнал экземпляра процесса, второе — личный идентификатор отправляющего сигнал экземпляра, остальные — собственно сигнал.

Переход $remove$ будет возможен, если значение третьего поля первого элемента списка, соответствующего очереди к экземпляру процесса с личным идентификатором p, равно $kill$. Все остальные условия в спусковой функции определяют изъятие из мест $State$, $self$, $sender$, $parent$ и $offspring$ по фишке, значение первого поля которых совпадает с p.

При срабатывании перехода $remove$ из входных мест $State$, $sender$, $self$, $parent$ и $stop$ заберется по одной фишке, соответствующей экземпляру процесса с личным идентификатором p, первый элемент списка (p, h, s) изымается из места $queue$, в месте $stop$ фишка значения $(p, 1)$ заменится на фишку значения $(p, 0)$.

8. МОДЕЛИРОВАНИЕ СРЕДСТВ УПРАВЛЕНИЯ ВРЕМЕНЕМ

Комплекс, реализующий систему SDL, должен обладать часами с абсолютным временем в согласованных единицах. Измерение отрезков времени осуществляется функцией now. Оператор $set(N, t)$ устанавливает в таймере t время N в принятых единицах, причем новая установка таймера отменяет предыдущую. Таймер считается активным с момента его установки и до момента восприятия процессом от него сигнала. Таймер можно перевести в неактивное состояние (говорят "сбросить таймер") оператором $reset$. С момента инициации системы и до первой установки таймера он также считается неактивным.

Предположим, что в описании некоторого процесса имеется переход, содержащий оператор установки таймера — $set(N, timer)$. Рассмотрим основные моменты моделирования этого перехода. По описанию перехода будет построена сеть такая, как показано на рис. 9. Различные экземпляры этого процесса будут моделироваться одной сетью, но первое поле каждой фишки в сети будет указывать, какому экземпляру процесса она соответствует.

Прежде чем приступить к описанию этой сети, построим подсеть, назовем ее $save$. В процессе ее функционирования моделируется удаление из очереди некоторого экземпляра процесса сигнала, находящегося в этой очереди быть может не первым.

8.1. Описание подсети *save*

На рис. 8 подсеть *save* моделирует сохранение всех сигналов в очереди некоторого экземпляра процесса, кроме одного — сигнала от таймера.

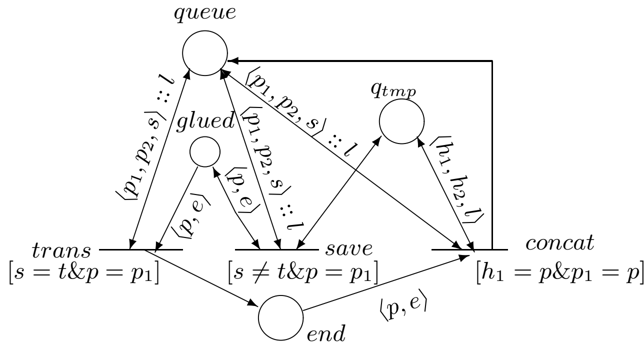

Рис. 8. Подсеть *save*

Подсеть имеет служебное место *glued*, фишка в нем может принимать значение из множества цветов *pair*. Значение первого поля записи этой фишки есть личный идентификатор экземпляра процесса, для которого реализуется удаление сигнала от таймера из его очереди, значение второго поля — e. Подсеть содержит служебное место q_{tmp}, которое может иметь фишку типа *list* и предназначается для хранения сигналов, находящихся в очереди этого экземпляра процесса до сигнала от таймера. Изначально это место имеет значение *nil*.

В подсети создано три перехода: *trans*, *save* и *concat*. Спусковые функции первых двух переходов определяются сигналом от таймера t и номером экземпляра процесса, для которого реализуется удаление сигнала от таймера из его очереди.

Как только в месте *glued* появится фишка, обозначим ее за (p, e), а в месте *queue* будет содержаться фишка, моделирующая очередь сигналов к экземпляру процесса с личным идентификатором p, в подсети станет возможным либо переход *trans*, либо переход *save*. Если первым в очереди сигналов к экземпляру процесса с номером p будет находиться сигнал от таймера, то может сработать переход *trans*, иначе — *save*. При срабатывании перехода *save* первый элемент фишки, моделирующей очередь сигналов к этому экземпляру процесса, изымается

и добавляется к списку в месте q_{tmp}, а из места $glued$ забирается фишка значения (p, e), она же в него и возвращается. Переход $save$ может срабатывать до тех пор, пока первым в очереди к этому экземпляру не окажется сигнал от таймера t. После нескольких срабатываний перехода $save$ фишка в месте $queue$, моделирующая очередь сигналов к экземпляру процесса с номером p, будет представлять хвост первоначальной очереди — элементы, моделирующие сигналы в очереди, находящиеся после сигнала от таймера t.

При срабатывании перехода $trans$ из места $glued$ заберется фишка значения (p, e) и поместится в место end, в результате чего станет возможным переход $concat$. Спусковая функция этого перехода зависит от номера p. При срабатывании этого перехода из места $queue$ заберется фишка, представляющая список и моделирующая хвост очереди к экземпляру процесса с номером p, из места q_{tmp} заберется фишка, моделирующая начало очереди к этому экземпляру процесса, и осуществится конкатенация двух списков. Новая фишка, представляющая конкатенацию двух списков, поместится в место $queue$. Таким способом функционирование подсети $save$ моделирует изъятие из очереди некоторого экземпляра процесса сигнала от таймера.

8.2. Сетевое представление оператора *set*

Сеть, соответствующая SDL-переходу, содержащему оператор установки таймера, представлена на рис. 9. Она имеет служебные переходы $begin$ и end и подсети $net1$ и $net2$, моделирующие операторы, соответственно находящиеся до и после оператора set в SDL-переходе. В обведенном штриховой линией прямоугольнике находится подсеть $save$ (см. рис. 8). Чтобы не загромождать рисунок, некоторые места и переходы, а также дуги подсети $save$ не изображены, а показаны только новые дуги, которые соединяют элементы этой подсети с другими элементами.

В сети создано место now, которое содержит одну фишку, несущую временной штамп. В процессе функционирования моделирующей сети временной штамп фишки в месте now будет определять текущий момент в системе.

Переход add имеет временную пометку 1. Место now будет входным и выходным местом этого перехода. Никаких других входных и выходных мест переход add не имеет. При срабатывании этого перехода фишка значения n с временным штампом, равным значению текущего времени в системе, изымается из места now, и помещается в него фишка

значения $n+1$ с временным штампом на единицу больше предыдущего. Следовательно, фишка в месте *now* станет доступна переходу *add* через одну единицу времени.

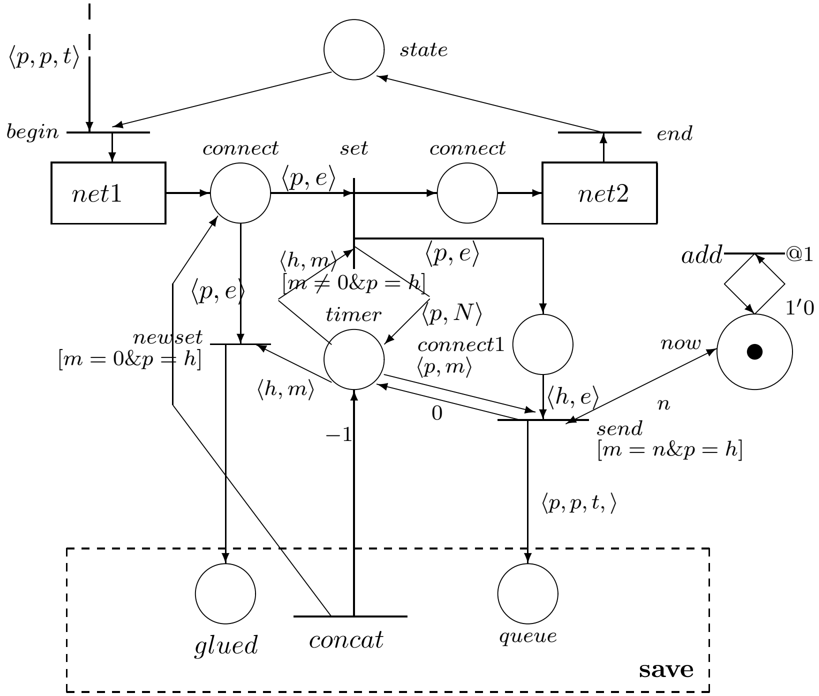

Рис. 9. Сеть для перехода, содержащего оператор *set*

Для отображения таймера t в сети создано место *timer*, фишки в нем могут принимать значения из множества цветов *pair*. Если экземпляр процесса создается в момент инициации системы, то начальная разметка этого места — одна фишка значения $(p, -1)$, где p — личный идентификатор этого экземпляра. Фишка такого цвета будет означать, что таймер у этого экземпляра процесса находится в неактивном состоянии, фишка цвета $(p, 0)$ будет означать, что сигнал от таймера находится в очереди у этого экземпляра процесса, фишка любого другого цвета будет означать, что произошла установка таймера, но значение

таймера еще больше, чем текущее время в модели. Для экземпляра процесса, создающегося в процессе функционирования системы, в место *timer* фишка значения $(p, -1)$ поместится после срабатывания перехода *create*.

Служебное место *connect*1 соединяет переход *set* с переходом *send*, моделирующим посылку сигнала от таймера любого экземпляра процесса в очередь к себе. Переход *send* может сработать в том случае, если значение второго поля фишки (p, m) в месте *timer* совпадает со значением фишки в месте *now*. Это соответствует тому, что значение в таймере у экземпляра процесса, которому при моделировании соответствует фишка (p, m), стало равно текущему времени в системе. Если в месте *timer* имеется N различных фишек, моделирующих различные экземпляры одного процесса, у каждой из которых значение второго поля совпадает со значением фишки в месте *now*, то переход *send* должен сработать N раз. Здесь рассматривается модель, в которой допускается кратная возможность переходов, т. е. в сети имеются переходы, у которых во входных местах в любой момент времени имеется несколько фишек, которых достаточно для двух или более одновременных срабатываний. Таким образом переход *send* сработает N раз в системе в момент времени, определенный временным штампом фишки в месте *now*.

Срабатывание перехода *send* заменит фишку значения (p, m) в месте *timer* на фишку значения $(p, 0)$ и добавит элемент (p, p, t) к фишке в месте *queue*, моделирующей очередь к экземпляру процесса с личным идентификатором p. Таким образом, фишка значения $(p, 0)$ в месте *timer* означает, что сигнал от таймера от экземпляра процесса с номером p, находится в очереди процесса.

Потребление сигнала от таймера каким-либо простым SDL-переходом этого экземпляра процесса моделируется срабатыванием моделирующего перехода в сети, которое изымет первый элемент из списка в месте *queue*, соответствующий сигналу от таймера этого экземпляра, и заменит 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 установку.

При этой отмене возможны два варианта. Они зависят от того, был ли послан в очередь экземпляра процесса сигнал от таймера при предыдущей установке, т.е. достигло ли текущее время в системе значения, заданного предыдущей установкой таймера. Если сигнал от таймера еще не был послан в очередь экземпляра процесса, то новая установка таймера должна изменить предыдущее значение таймера, иначе сигнал от предыдущей установки таймера должен быть удален из очереди сигналов этого экземпляра процесса.

В моделирующей сети это будет осуществляться следующим образом. Если значение второго поля фишки в месте $timer$ не равно 0 (соответствие того, что сигнал от таймера не в очереди), может сработать переход set. В результате его срабатывания фишка в месте $timer$ заменится на новую, соответствующую новому установленному времени. Иначе (см. рис. 9, $m = 0$) в сети может сработать переход $newset$. После срабатывания перехода $newset$ возможно срабатывание переходов сети $save$, 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.

9. ЗАКЛЮЧЕНИЕ

В работе описана процедура трансляции SDL-спецификаций с динамическими конструкциями в раскрашенные сети Йенсена, обогащенные приоритетами. Расширения этих сетей приоритетами естественно и позволяет развить средства симуляции и анализа.

Способ моделирования основан на том, что в многоуровневом описании системы в SDL позиция каждого экземпляра процесса в общей иерархии системы остается неизменной, что позволяет описание системы транслировать в структуру сети, а экземпляры процесса моделировать с помощью фишек.

В результате работы алгоритма создается такая сетевая модель, в которой в каждом месте будет содержаться не более одной фишки, моделирующей некоторый экземпляр процесса. Таким образом, если во время функционирования системы может существовать n различных экземпляров какого-либо процесса, то в каждом месте моделирующей его сети может содержаться не более n фишек, причем каждая фишка будет соответствовать своему экземпляру процесса. Это факт позволя-

ет существенно повысить эффективность моделирования, так как существенно уменьшает перебор вариантов связывания переменных. Заметим, что хотя доказательств в этой работе не приводится, однако на содержательном уровне поясняется, как семантика конструкций языка SDL представляется в формальной семантике раскрашенных сетей Йенсена.

Данная работа является частью большого проекта, цель которого — валидация коммуникационных протоколов. Проект содержит в себе две части. Первая предназначена для автоматического построения сетевой модели с использованием описанной в данной работе процедуры трансляции, вторая — система NetCalc — позволяет редактировать и симулировать построенную сетевую модель.

Автор выражает благодарность В. А. Непомнящему за постоянное внимание и поддержку работы, Е. В. Окунишниковой за совместную плодотворную разработку алгоритма трансляции Estelle-спецификаций в раскрашенные сети Петри, идеи которого использованы в этой работе.

СПИСОК ЛИТЕРАТУРЫ

1. **Richier J. L., Rodriguez C., Sifakis J., Voiron J.** Verification in XESAR of the Sliding Window protocol // Proc. IFIP Intern. Sympos. on Protocol Specification, Testing and Verification VII. — Amsterdam: North-Holland, 1987. — P. 235—248.
2. **Dimitrov V., Petkov A.** Verification oriented Estelle specification of communication protocols // Reseach into Networks and Distributed Applications. — Amsterdam: North-Holland, 1988. — P. 953—960.
3. **Lai R., Jirachiefpattana A.** Verifying Estelle protocol specifications using numerical Petri nets // Comput. Syst. Sci & Eng. — 1996. — Vol. 11, N. 1. — P. 15—33.
4. Верификация Estelle-спецификаций распределенных систем посредством раскрашенных сетей Петри /В.А. Непомнящий, А.В. Быстров и др. — Новосибирск, 1998. — 140 с.
5. **Bause F. et al.** SDL and Petri net perfomance analysis of communicating systems // Proc. IFIP 15th Intern. Symp. on Protocol Specification, Testing and Verification. — Warsaw, Poland, 1995. — P. 259—272.
6. **Fisher J., Dimitrov E.** Verification of SDL'92 specifications using extended Petri nets // Proc. IFIP 15th Intern. Conf. on Protocol Specification, Testing and Verification. — Warsaw, Poland, 1995. — P. 455—458.
7. **Jensen K.** Coloured Petri nets: Basic concepts, analysis methods and practical use. — Berlin a. o.: Springer-Verlag, 1996. — Vol. 1. Basic concepts.
8. **Jensen K.** Coloured Petri nets: Basic concepts, analysis methods and practical use. — Berlin a. o.: Springer-Verlag, 1996. — Vol. 2. Analysis methods.
9. **Jensen K.** Coloured Petri nets: Basic concepts, analysis methods and practical use. — Berlin a. o.: Springer-Verlag, 1997. — Vol. 3. Practical use.
10. **Чурина Т.Г.** Способ построения раскрашенных сетей Петри, моделирующих SDL-системы. — Новосибирск, 1998. — 56 с. — (Препр. / СО РАН. ИСИ; N 56).

11. **Котов В. Е.** Сети Петри. — М.: Наука, 1984.
12. **Recommendation** Z.100 CCITT Specification and Description Language (SDL)
13. **Карабегов А.В., Тер-Микаэлян Т.М.** Введение в язык SDL.— М.: Радио и связь, 1993.
14. Использование сетей Петри для верификации распределенных систем, представленных на языке Estelle. //В.А. Непомнящий, А.В. Быстров и др. — Известия Академии наук, серия: Теория и системы управления, 1999, N 5.
15. **Churina T. G., Okunishnikova E. V.** Coloured Petri nets approach to the validation of Estelle specifications // Proc. of Workshop on Concurrency, Specification and Programming. — Warsaw, Poland, 1997. — P. 25—36.
16. **Churina T. G., Okunishnikova E. V.** Modelling Estelle specifications using coloured Petri nets //Joint Bulletin of NCC and IIS. — Novosibirsk, issue 8 1998. — P. 19—38.

Т. Г. Чурина

МОДЕЛИРОВАНИЕ ДИНАМИЧЕСКИХ КОНСТРУКЦИЙ ЯЗЫКА SDL ПОСРЕДСТВОМ РАСКРАШЕННЫХ СЕТЕЙ ПЕТРИ

Препринт
71

Рукопись поступила в редакцию 14.12.1999

Рецензент Н. В. Шилов

Редактор З. В. Скок

Подписано в печать 31.01.2000

Формат бумаги 60×84 1/16 Объем 2,1 уч.-изд.л., 2,2 п.л.

Тираж 50 экз.

ЗАО РИЦ "Прайс-курьер", 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6