УДК 004.7

В. А. Непомнящий, Н. С. Попова, Т. Г. Чурина

Институт систем информатики им. А. П. Ершова СО РАН
пр. Акад. Лаврентьева, 6, Новосибирск, 630090, Россия

Новосибирский государственный университет
ул. Пирогова, 2, Новосибирск, 630090, Россия

E-mail: vnep@iis.nsk.su; popova.natalya@gmail.com; tanch@iis.nsk.su

МОДЕЛИРОВАНИЕ СПЕЦИФИКАЦИЙ РАСПРЕДЕЛЕННЫХ СИСТЕМ НА ЯЗЫКЕ DYNAMIC-REAL СЕТЯМИ ПЕТРИ ВЫСОКОГО УРОВНЯ

Рассматриваются спецификации распределенных систем на языке Dynamic-REAL (dREAL) с динамическим порождением и уничтожением экземпляров процессов. В качестве сетевой модели выбраны модифицированные раскрашенные сети Петри – иерархические временные типизированные сети (ИВТ-сети), в которых используются приоритеты, специальные места, представляющие очереди фишек, и концепция интервального времени. Предложен метод трансляции языка dREAL в ИВТ-сети. На основе этого метода реализован транслятор из языка dREAL в эту сетевую модель.

Ключевые слова: распределенные системы, язык Dynamic-REAL, раскрашенные сети Петри, иерархические временные типизированные сети, метод трансляции.

Введение

В последние годы заметно возросла роль формальных методов, применяемых для разработки распределенных и телекоммуникационных систем. Это связано с тем, что для них усложняется документирование, анализ и верификация. Для преодоления указанных трудностей на практике используется язык выполнимых спецификаций SDL, принятый в качестве стандарта международной организацией ITU (International Telecommunication Union) [1; 2]. Верификация выполнимых спецификаций, представленных на языке SDL, заключается в проверке корректности их ключевых свойств и является актуальной задачей современного программирования.

С целью упрощения анализа и верификации SDL-спецификаций был разработан язык выполнимых спецификаций Basic-REAL, базирующийся на статическом подмножестве языка SDL [3; 4]. Преимущества языка Basic-REAL обусловлены простым синтаксисом, допускающим графические представления выполнимых спецификаций, и полной компактной операционной семантикой. В работе [5] описан язык Dynamic-REAL (dREAL), полученный расширением языка Basic-REAL посредством динамических конструкций порождения и уничтожения экземпляров процессов. В работе [6] представлен программный комплекс SRDSV (SDL/REAL Distributed Systems Verifier) для моделирования, анализа и верификации SDL-спецификаций распределенных систем. Этот комплекс включает транслятор из языка SDL в язык dREAL, систему автоматического моделирования dREAL-спецификаций и две системы верификации этих спецификаций методом проверки моделей. Этот программный комплекс был успешно применен для анализа и верификации динамической системы управления сетью касс-терминалов.

Верификация распределенных систем значительно упрощается во многих случаях при использовании таких моделей, как конечные автоматы, сети Петри и их обобщения. Среди этих моделей можно выделить раскрашенные сети Петри (РСП) [7], так как они имеют значитель-

ную выразительную силу, большой опыт применения и для них реализованы мощные средства анализа [8; 9]. Однако ручное построение моделей распределенных систем часто приводит к ошибкам. Поэтому возникает задача автоматического построения сетевых моделей для распределенных систем, представленных на языке SDL. Проблема автоматического перевода SDL-спецификаций в РСП была указана в [7] как открытая проблема. Трансляция подмножества SDL в так называемые SDL-сети, расширяющие стандартные сети Петри временными интервалами и охранными условиями для переходов, описана в [10]. Трансляция из SDL в сети Петри высокого уровня, названные M-сетями, представлена в [11]. Практический метод трансляции из SDL в так называемые Pr/T-сети рассмотрен в [12]. Заметим, что все перечисленные сетевые модели существенно отличаются от РСП.

Таким образом, возникает задача автоматического перевода dREAL-спецификаций в сети Петри высокого уровня, поскольку во многих случаях верификация сетевых моделей dREAL-спецификаций упрощается по сравнению с верификацией сетевых моделей SDL-спецификаций. В качестве эффективной сетевой модели выбрана модификация РСП – иерархические временные типизированные сети (ИВТ-сети), в которых используются приоритеты, специальные места, представляющие очереди фишек, и концепция интервального времени [13]. Эта задача и рассматривается в данной работе.

Обзор языка dREAL

Язык dREAL состоит из языка выполнимых спецификаций для представления распределенных систем, который рассматривается в данной работе, и языка логических спецификаций для представления их свойств. Система в языке dREAL состоит из блоков, соединенных между собой и с окружающей средой каналами. Средствами языка dREAL обеспечивается многоуровневое описание системы. Блок может содержать другие блоки и процессы, которые работают параллельно и взаимодействуют друг с другом и с внешней средой посредством обмена сигналами через однонаправленные каналы. С каждым действием процесса ассоциируется временной интервал, который задает продолжительность выполнения действия. Каждый экземпляр процесса может порождать или уничтожать экземпляры процессов в своем блоке. Содержимым каналов являются сигналы, возможно с параметрами. Каналы имеют структуру очереди. Время жизни сигнала с параметрами в канале ограничено одним запросом на чтение. Процесс языка dREAL описывает последовательность таких действий, как изменение переменных, чтение сигналов из каналов, запись сигналов в каналы, очистка каналов.

Язык dREAL подробно рассмотрен в [5], где описывается его синтаксис и формальная семантика. В данной работе представлена версия языка dREAL, которая описана в [6]. В этой версии dREAL имеются определенные ограничения, связанные с использованием этой версии в качестве входного языка программного комплекса SRDSV, например, простое интервальное время.

Сетевая модель

Раскрашенные сети Петри [7] являются расширением ординарных сетей Петри. *Ординарную* сеть Петри можно определить как ориентированный граф с вершинами двух типов: *местами* и *переходами*, соединенными дугами таким образом, что каждая дуга соединяет вершины различных типов. В каждом месте сети может содержаться неотрицательное количество фишек, называемое его разметкой. Разметка сети – это совокупность разметок всех мест сети.

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.

В отличие от ординарных сетей Петри каждая фишка в раскрашенной сети обладает индивидуальностью – значением некоторого типа, которое называется *цветом*.

В ИВТ-сети различаются два вида мест – обычные и *многослойные*. Многослойное место представляет собой запись, состоящую из двух полей. В первом поле содержится целое число, которое будем называть *номером слоя*. Во втором поле, называемым *контентом*, – значение любого допустимого типа. Две фишки *принадлежат* одному слою, если значения первых полей этих фишек совпадают. Заметим, что все фишки, хранящиеся в очереди, принадлежат одному слою. Многослойное место, которое не является очередью, может содержать не более одной фишки, принадлежащей одному слою. Многослойное место-очередь может содержать не более одной очереди в каждом слое.

В отличие от ординарных сетей возможность срабатывания перехода в ИВТ-сети зависит не только от наличия фишек во входных местах перехода, но и от их значений. ИВТ-сети являются *квазибезопасными* в том смысле, что каждое место может иметь не более одной доступной фишки, принадлежащей определенному слою.

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

Поведение иерархической сети эквивалентно поведению неиерархической сети, получающейся при замещении всех модулей страницами, которые они представляют.

В ИВТ-сетях, как и в языке dREAL, используется интервальное время. ИВТ-сети имеют приоритеты: из нескольких возможных переходов срабатывает любой, приоритет которого не меньше, чем у остальных возможных переходов.

Перевод dREAL-спецификаций в ИВТ-сети

Сеть, моделирующая dREAL-спецификацию, строится с помощью поэтапного уточнения. На первом этапе создается страница, которая соответствует основной структуре спецификации и содержит по одному модулю для каждого описания блока и процесса. На втором этапе для каждого из полученных модулей строится дерево страниц, повторяющее иерархию dREAL-блока с корнем в соответствующем модуле. На следующих этапах создаются сети, соответствующие процессам, затем – dREAL-переходам.

Для представления стандартных типов целый, булевский и вещественный используются соответствующие множества цветов: *integer*, *boolean* и *real*. Для определения структурированных типов данных допускается использование массивов и записей. Записи в моделирующей сети представляются множествами цветов, полученных с помощью декларативной приставки *product*.

При отображении dREAL-спецификации считаем, что все декларации вынесены на отдельную страницу. Определения новых типов преобразуются в множества цветов. 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. Фишка со значением *e* называется *ординарной* или *бесцветной* и используется в сетях в служебных целях.

Моделирование структуры REAL-спецификации. Строящаяся сеть содержит по одному модулю на каждый блок, описанный в программе на dREAL. В качестве имени модуля для большей наглядности будем использовать имя соответствующего блока.

Каждый канал в dREAL имеет ассоциированную с ним FIFO-очередь, в которой сохраняются сигналы, полученные через этот канал. При трансляции dREAL-спецификации каналы

представляются местами специального типа – очередями, а сигналы, сохраняемые в очередях, – фишками. При начальной разметке все места, соответствующие каналам, пусты. Каждый канал в сети представляется одним местом. Входной канал отображается местом, являющимся входным для модуля, представляющего блок, который связан с этим каналом, выходной канал – выходным местом для этого модуля.

Описания всех сигналов, которые могут передаваться по каналам в спецификации, переходят в декларации сети. При этом создаются множества цветов, которые будут использоваться при моделировании взаимодействий между блоками. Так как выходные сигналы посылаются либо всем экземплярам процесса, либо процессу с определенным ПИД, то в каналы передаются сообщения, которые формируются двумя способами и состоят из следующих элементов: ПИД экземпляра-получателя либо специальное значение *e*, указывающее, что сообщение предназначается любому экземпляру; сигнал, возможно с параметрами, указанный в операторе WRITE. Поэтому множество цветов, приписываемое месту-каналу, в общем случае формируется следующим образом:

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

где *sig_type* – множество цветов, полученное при отображении сигналов, передаваемых по каналу. Множество цветов *Receiver_Pid* позволяет формировать сообщения двух форматов. Фишка, принимающая значения из множества *Channel_type*, имеет вид (*id*, *sid*, *SigName*, *S*), где *id* – либо личный идентификатор экземпляра процесса-получателя, либо значение *e*; *sid* – ПИД процесса-отправителя; *SigName* – имя сигнала; *S* – сигнал.

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 конструкции порождения других процессов.

По завершению первого шага моделирования получаем сеть, состоящую из модулей, соответствующих блокам в спецификации и мест-очередей, соответствующих каналам.

Рассмотрим спецификацию ExampleProtocol, содержащую описание блока, в котором содержится три процесса B1, B2, B3 и три канала C1, C2 и C3. Каналы C1 и C2 соединяют процессы B1 и B2, канал C3 соединяет процесс B2 с окружением. По каналам C1, C2 и C3 передаются сигналы s1, s2 и s3 соответственно.

```
ExampleProtocol: BLOCK
INN UNB QUEUE CHN C1          // Объявляется канал C1
FOR s1.                       // по нему может передаваться сигнал s1
INN UNB QUEUE CHN C2
FOR s2.
OUT UNB QUEUE CHN C3
FOR s3.
FROM B1 CHN C1 TO B2. // Какие процессы связывает канал C1
FROM B1 CHN C2 TO B2.
FROM B2 CHN C3 TO ENV.
B1: BLOCK
…
END; /* B1 */
B2: BLOCK
…
END; /* B2 */
END; /* ExampleProtocol*/
```

Для этой спецификации построена сеть, показанная на рис. 1. Чтобы не загромождать рисунок, декларация сети опущена.

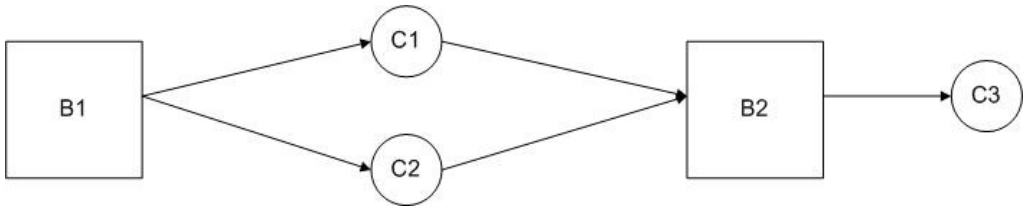

Рис. 1. Верхняя страница сети для протокола ExampleProtocol

Моделирование блока. При разбиении блока возникает структура, аналогичная структуре спецификации, в которой блок играет роль внешней среды. Блок в языке dREAL может в свою очередь состоять из подблоков или процессов и содержать *внутренние* каналы. На каждой странице, связанной с модулем, соответствующим некоторому блоку, отображается внутренняя структура этого блока. Это отображение осуществляется таким же способом, что и отображение структуры спецификации на первой странице. Каждому подблоку на подстранице соответствует один модуль, каждому внутреннему каналу – одно место.

Для порождения экземпляров процессов на этом шаге создаются дополнительные места *create_pr*, где *pr* – имя порождаемого процесса. Каждое из мест *create_pr* является входным для модуля, соответствующего порождаемому процессу с именем *pr*, и выходным для модуля, представляющего «родительский» процесс. При этом если некий процесс в своих переходах содержит *n* операторов CREATE и каждый из них имеет вид либо CREATE *A*, либо CREATE *B*, то в сети будет создано два места *create_A* и *create_B*, которые будут выходными для модуля, соответствующего этому процессу. Фишки в месте *create_pr* принимают значения *integer* и соответствуют ПИД создаваемого процесса.

Поскольку понятие «переход» определено и для языка dREAL, и для ИВТ-сетей, то далее в тексте для обозначения переходов в моделирующей сети будем использовать термин *N-переход* в тех случаях, где значение неясно из контекста. Кроме того, элементы сети будем выделять курсивом, а dREAL-программы будем писать разреженным шрифтом.

Моделирование процесса. В процессе функционирования системы количество экземпляров процессов может изменяться, не превышая некоторого числа, заданного для процесса, но его позиция в общей иерархии остается неизменной. Описание процесса будем моделировать структурой сети, а экземпляры – наборами фишек в местах этой сети. При этом несколько фишек, относящихся к различным экземплярам, могут находиться в одном месте. Различаются они по первому полю, которое имеет значение ПИД этого экземпляра. Таким образом, фишки, принадлежащие конкретному экземпляру процесса, будут относиться к одному слою сети.

Каждому dREAL-переходу на этом этапе соответствует либо модуль либо, один *N*-переход. Операторам очистки каналов, порождения и уничтожения процессов соответствуют модули, остальным операторам – *N*-переходы. Каждой описанной в процессе переменной сопоставляется одно многослойное место. Начальная разметка второго поля места-переменной соответствует начальному значению переменной, если оно определено, или нулю, если не определено.

Кроме того, на подстранице, соответствующей процессу, присутствуют многослойные места *self*, *state*, *count*, места-очереди, соответствующие всем входным каналам этого процесса, и модуль *channel_planner.* Служебное место *self* соответствует переменной SELF в dREAL, начальная разметка – номер, который присваивается экземпляру процесса после создания сети. Множество состояний процесса определяет множество цветов второго поля мес-

та *state*. С его помощью обеспечивается неделимость выполнения dREAL-перехода. Служебное место *count* содержит фишку с целым значением, равным количеству созданных экземпляров моделируемого процесса. Значение этой фишки изменяется при создании нового экземпляра и при уничтожении существующего.

Доставка сигналов в языке dREAL возможна как всем экземплярам процесса, так и конкретному экземпляру. Реализуют эту возможность специальный модуль *channel_planner* (рис. 2), который будет описан ниже, и место, соответствующее входному каналу этого процесса.

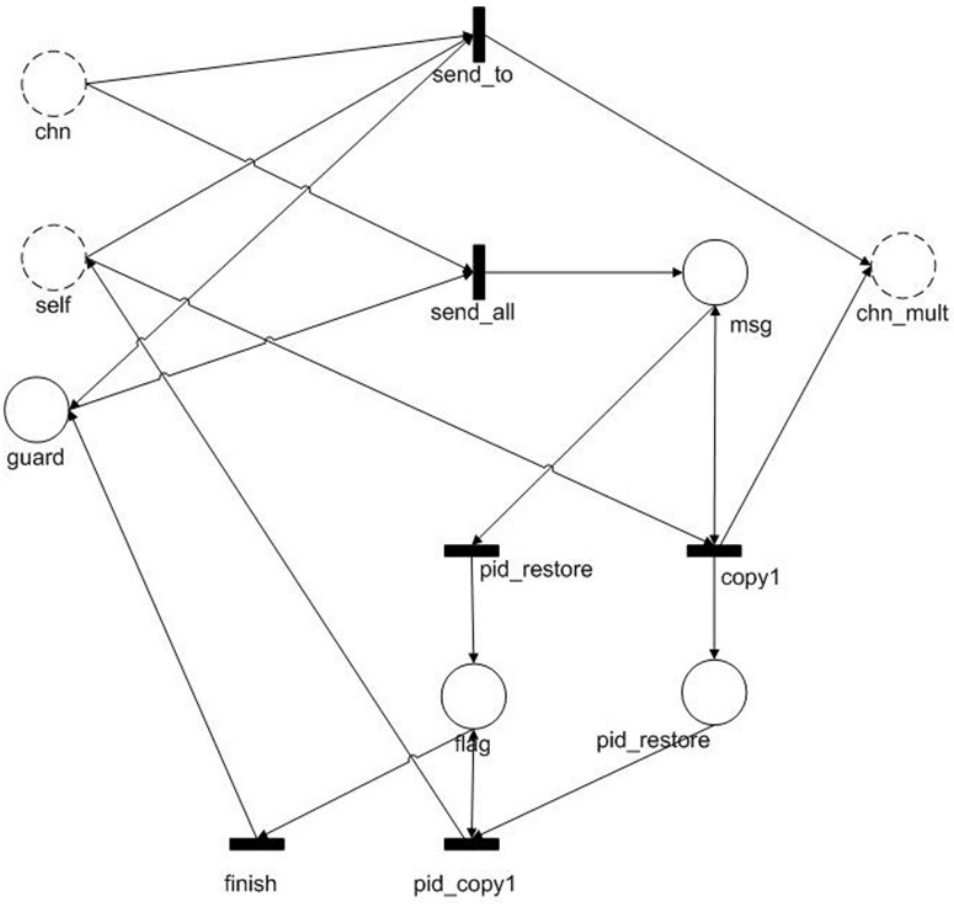

Рис. 2 Модуль chn_planner

Функционирование процесса в dREAL начинается с выполнения первого перехода. Поскольку в языке dREAL несколько переходов могут находиться в одном и том же состоянии, то далее происходит недетерменированный выбор следующего для исполнения dREAL-перехода.

Моделирование перехода. Спусковую функцию *N*-перехода, моделирующего dREAL-переход, образуют состояние, указанное в спецификации после слова TRANSITION, имена входных сигналов и условное выражение, соответствующее условному выражению dREAL-перехода. Содержимое *N*-перехода определяется операторами, составляющими тело dREAL-перехода. Если некоторая переменная используется в переходе процесса, то место, соответствующее данной переменной, будет входным и выходным для *N*-перехода, представляющего этот dREAL-переход. Если в dREAL-переходе процесса осуществляется посылка сигнала,

то добавляется дуга от соответствующего *N*-перехода к месту, моделирующему выходной канал процесса.

В том случае когда dREAL-переход представляется в сети модулем, на соответствующей ему странице присутствуют копии всех мест, которые связаны с этим модулем. Входные / выходные дуги повторяют соединение мест-прототипов с модулем. Место *state* будет входным и выходным для *N*-перехода, моделирующего первый dREAL-переход. Выражение на выходной дуге перехода, соединяющей этот переход с местом *state*, определяется состоянием, следующим в процессе за первым переходом. Ниже представим отображение некоторых основных операторов языка, таких как посылка сигнала, порождение и уничтожение экземпляров процесса, очистка каналов, каждый из которых моделируются подсетью.

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

Моделирование порождения и уничтожения процесса. Как было отмечено, на этапе моделирования блока создавались дополнительные места-очереди *cr_pr*. Каждое место *cr_pr* является входным для модуля, соответствующего порождаемому процессу с именем *pr*, и выходным для модуля, представляющего родительский процесс. Каждая фишка в месте *cr_ pr* есть запись, где значение первого поля есть личный идентификатор создаваемого оператором CREATE экземпляра процесса, значение второго поля – личный идентификатор «экземпляра-родителя».

Оператор CREATE в сети представляется тремя переходами: *generate*, *generate_nul* и *create*. Переходы *generate* и *generate_nul* принадлежат сети, соответствующей «родителю», для них выходным будет место *create_pr*, а место *PId* – входным и выходным. Переход *create* располагается в сети порождаемого процесса, и для него служебные места *self*, *State* 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 *create_process* добавляется фишка, значение которой содержит личный идентификатор порождаемого экземпляра процесса. При срабатывании перехода *create* в сети создается новый слой, который представляет порожденный экземпляр процесса. Это осуществляется путем добавления в каждое многослойное место по одной фишке, значение первого поля которой представляет ПИД этого экземпляра.

При моделировании уничтожения процесса в подсети процесса создаются *N*-переходы с именами *stop* и *cln_var*. Места *State* и *self* являются входными для *N*-перехода *stop*, а все места, соответствующие переменным dREAL-программы, – входными для *N*-перехода *cln_var*. При срабатывании *N*-перехода *stop* из мест *self* и *State* изымаются фишки, а при срабатывании *N*-перехода *cln_var* изымаются фишки из мест-переменных. В результате в сети в слое, соответствовавшем экземпляру процесса, не остается фишек. Таким образом, данный экземпляр перестает существовать.

Моделирование доставки сигналов экземплярам процессов. Как было отмечено в разделе моделирование процессов, с помощью модуля *chn_planner* реализуется доставка сигнала из канала нужному экземпляру-получателю либо сигнал дублируется и доставляется каждому экземпляру процесса. Напомним, что входным местом этого модуля является однослойное место *chn* (см. рис. 2), моделирующее входной канал, а выходным – многослойное место *chn_mult*, содержащее очереди сигналов ко всем экземплярам данного процесса.

Для моделирования доставки сигналов в сети создается *N*-переход *send_to* с приоритетом 4. Соединение этого перехода с местами показано на рис. 2, спусковая функция – есть выражение, проверяющее соответствие ПИД экземпляра-получателя и значения фишки в месте *self*. При функционировании подсети *chn_planner* сигнал добавляется в место *chn_mult* в конец очереди каждого слоя. Это осуществляется при срабатывании *N*-перехода *send_all* с приоритетом 4. Спусковой функцией перехода *send_all* является выражение, проверяющее,

что сигнал предназначается всем экземплярам процесса. С помощью места *guard* гарантируется, что новое срабатывание перехода *send_all* не осуществится до тех пор, пока первый сигнал из места-очереди *chn* не добавится в очередь каждого экземпляра процесса.

При срабатывании перехода *send_all* первый сигнал забирается из очереди места *chn* и помещается в место *msg*. В место *count_copy* помещается фишка со значением, равным количеству функционирующих экземпляров процесса. Переход *copy*1 сработает столько раз, сколько фишек имеется в месте *self*. При его срабатывании из места *msg* забирается фишка-сигнал, она же в него и возвращается, а из места *self* забирается фишка с номером экземпляра процесса *n* и помещается в место *pid_copy* в слой с номером *n*.

Как только в месте *self* не останется фишек, сработает переход *pid_restore*. При его срабатывании фишка изымается из места *msg* и помещается в место *flag*, после чего становится возможным переход *pid_copy*1. Срабатывание последнего состоит в следующем: изымается фишка из места *flag* и добавляется в место *chn_mult* в слой с тем же номером; в место *self* помещается номер экземпляра процесса, в очередь которого сообщение добавилось.

Переход *pid*_copy1 сработает столько раз, сколько существует экземпляров процесса. После этого становится возможным переход *finish*. Его срабатывание изымает фишку из места *flag* и помещает фишку в место *guard*, что делает возможным новое срабатывание перехода *send_all*.

Моделирование очистки каналов. В языке dREAL выполняется как очистка входного и выходного каналов определенного экземпляра процесса, так и всех экземпляров этого процесса.

Рассмотрим два описания процессов *A* и *B* и канал между ними *chn*, который является выходным для процесса *A* и входным для процесса *B*. В сети каналу соответствуют места *chn* и *chn_mult* (рис. 3). Место *chn_mult* находится внутри модуля, соответствующего процессу *B*. «Запросы» на очистку канала *chn* будут помещаться в место *chn_start_clean.* С помощью места *guard* гарантируется, что очередная очистка канала не начнется до тех пор, пока не закончится предыдущая. В месте *cln_request* будет содержаться информация о том, для какого экземпляра требуется очистить канал.

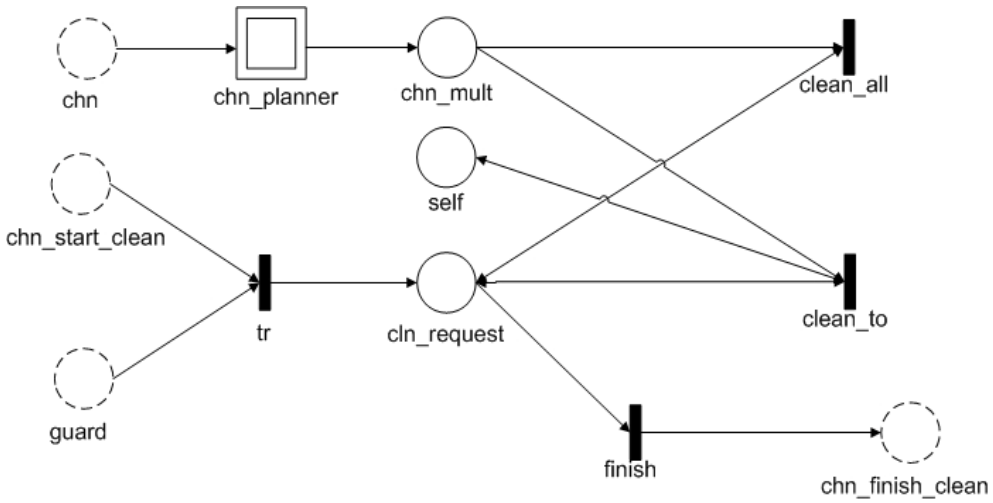

Рис. 3. Подсеть, моделирующая очистку канала

При функционировании *N*-перехода *tr* из места *guard* фишка изымается; из места *chn_start_clean* изымается фишка и помещается в место *cln_request*. *N*-переход *clean_all* может сработать в том случае, если очистку канала требуется выполнить для всех экземпляров процесса. Приоритет данного *N*-перехода равен 1. *N*-переход *clean_all* может сработать в том

случае, если требуется очистить канал у конкретного экземпляра процесса. Приоритет данного *N*-перехода равен 1. *N*-переход *finish* может сработать в том случае, когда очистка каналов закончена. При его срабатывании из места *request* изымается фишка и помещается в место *chn_finish_clean*.

Предположим, что один из переходов экземпляра процесса *A* очищает канал *chn*. В этом случае в сети создаются дуги, ведущие от модуля *A*, соответствующего процессу *A*, к местам *chn_start_clean* и *guard.* Также в сети создается дуга от места *chn_finish_clean* к модулю *A*.

В подсеть, соответствующую процессу *A*, добавляется *N*-переход с таким же именем, как и имя перехода в dREAL-программе, в котором происходит очистка канала. Для этого *N*-перехода место *State* будет входным, а места *chn_start_clean* и *self* – выходными. При срабатывании этого перехода в место *chn_start_clean* помещается фишка, содержащая ПИД экземпляра, для которого требуется очистить канал, и ПИД экземпляра процесса, который посылает запрос на очистку.

И, наконец, в сети создается *N*-переход *tr_end*, для которого места *State* и *guard* будут выходными, а места self и *chn_finish_clean* – входными. Данный переход может сработать, если значения фишек в местах *chn_finish_clean* и *self* совпадают. После срабатывания этого *N*-перехода в место *guard* возвращается фишка, после чего может начаться обработка следующего запроса на очистку канала.

Реализация и апробация алгоритмов моделирования

На основе описанных алгоритмов был реализован транслятор, позволяющий получить сетевую модель по входной dREAL-спецификации для дальнейшего ее анализа и валидации. Перевод dREAL-спецификации в сетевую модель проходит в два этапа. На первом этапе строится внутреннее представление спецификации в виде дерева разбора; на втором этапе генерируется сетевая модель.

Тестирование проводилось для нескольких протоколов, включая динамическую систему управления сетью касс-терминалов [6]. Данная система описывает работу десяти параллельно работающих касс по продаже билетов. Взаимодействие с пассажирами происходит следующим образом: пассажир подходит к кассе, выбирает нужную станцию, после этого, если у него достаточно денег, опускает их в монетоприемник, забирает билет и уходит. После этого касса готова к обслуживанию нового пассажира. Если денег на билет недостаточно, то пассажир уходит, а касса готова к обслуживанию следующего пассажира.

Заключение

В данной работе описаны алгоритмы трансляции dREAL-спецификаций в ИВТ-сети. Их реализация поддерживает динамические средства языка dREAL, которые позволяют изменять количество экземпляров процессов в процессе функционирования системы. Описанный в работе метод трансляции позволяет строить квазибезопасные сетевые модели языка dREAL, что упрощает анализ и верификацию распределенных систем.

В работе [13] описан программный комплекс SPV (SDL protocol verifier), который включает транслятор из языка SDL в ИВТ-сети, а также средства для редактирования, моделирования, визуализации и верификации этих сетевых моделей.

Несмотря на то, что язык dREAL ориентирован на верификацию выполнимых спецификаций, он, как и язык SDL, является универсальным языком значительной выразительной силы. Формальное обоснование трансляции с таких языков – известная открытая проблема. Общепринятым подходом к неформальному обоснованию алгоритмов трансляции является успешное тестирование транслятора, который реализует эти алгоритмы. Такое тестирование было проведено для транслятора, описанного в данной работе.

Наш подход имеет хорошие перспективы. Предполагается расширить комплекс SPV за счет интерфейса с известной системой верификации SPIN [14], использующей метод проверки моделей. Планируется также применить комплекс SPV к анализу и верификации dREAL-спецификаций распределенных систем из разных проблемных областей.

Список литературы

1. *Карабегов А. В., Тер-Микаэлян Т. М.* Введение в язык SDL. М.: Радио и связь, 1993.

2. *Grammes R., Gotzhein R.* SDL Profiles – Formal Semantics and Tool Support // Proc. FASE 2007. Lect. Notes in Comp. Sci. 2007. Vol. 4422. P. 200–214.

3. *Непомнящий В. А., Шилов Н. В., Бодин Е. В.* REAL: Язык для спецификации и верификации систем реального времени // Системная информатика. Новосибирск, 2000. Вып. 7. С. 174–223.

4. *Nepomniaschy V. A., Shilov N. V., Bodin E. V., Kozura V.E.* Basic-REAL: integrated approach for design, specification and verification of distributed systems // Proc. IFM 2002. Lect. Notes in Comp. Sci. 2002. Vol. 2335. P. 69–88.

5. *Непомнящий В. А., Бодин Е. В, Веретнов С. О.* Язык спецификаций распределенных систем Dynamic-REAL. Новосибирск, 2007. (Препр. / ИСИ СО РАН; № 147). URL: http://www.iis.nsk.su/preprints/pdf/147.pdf.

6. *Непомнящий В. А., Бодин Е. В., Веретнов С. О.* Моделирование и верификация распределенных систем, представленных на языке SDL, с помощью языка Dynamic-REAL. Новосибирск, 2010. (Препр. / ИСИ СО РАН; № 156). URL: http://www.iis.nsk.su/preprints/pdf/156.pdf.

7. *Jensen K.* Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Springer-Verlag, 1997. Vol. 1–3.

8. *Kristensen L. M., Christensen S., Jensen K.* The Practitioner's Guide to Coloured Petri Nets // Intern. J. on Software Tools for Technology Transfer. 1998. Vol. 2. No. 2. P. 98–132.

9. *Jensen K., Christensen S., Wells L.* Coloured Petri Nets and CPN Tools for Modeling and Validation of Concurrent Systems // Intern. J. on Software Tools for Technology Transfer. 2007. Vol. 9. P. 213–254.

10. *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.

11. *Fleischhack H., Grahlmann B.* A Compositional Petri Net Semantics for SDL // Lecture Notes in Computer Sci. 1998. Vol. 1420. P. 144–164.

12. *Aalto A., Husberg N., Varpaaniemi K.* Automatic Formal Model Generation and Analysis of SDL // Proc. SDL 2003. Lecture Notes in Computer Sci. 2003. Vol. 2708. P. 285–299.

13. *Непомнящий В. А., Аргиров В. С., Белоглазов Д. М., Быстров А. В., Четвертаков Е. А., Чурина Т. Г.* Моделирование и верификация коммуникационных протоколов, представленных на языке SDL с помощью сетей Петри высокого уровня // Программирование. 2008. № 6. С. 35–49.

14. *Holzmann G. J.* The SPIN Model Checker. Primer and Reference Manual. Addison-Wesley, 2004.

Материал поступил в редколлегию 31.05.2010

Nepomniaschy V. A., Popova N. S., Churina T. G.

MODELING DYNAMIC-REAL SPECIFIED DISTRIBUTED SYSTEMS BY HIGH LEVEL PETRI NETS

We consider distributed systems specified on the language Dynamic-REAL (dREAL) that includes dynamic constructs for generating and removing process instances. Modified coloured Petri nets called hierarchical timed typed nets (HTT-nets) are used as a net model for dREAL- specifications. The nets use priorities, the interval time concept and special places representing queues of tokens. A method for translation from the language dREAL into HTT-nets is described. Based on the method, a translator from the language dREAL into the net model has been implemented.

Keywords: distributed systems, language Dynamic-REAL, coloured Petri nets, hierarchical timed typed nets, translation method.