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

Н.В. Визовитин, В.А. Непомнящий

АЛГОРИТМЫ ТРАНСЛЯЦИИ UCM-СПЕЦИФИКАЦИЙ В РАСКРАШЕННЫЕ СЕТИ ПЕТРИ

Препринт
168

Новосибирск 2012

Описаны алгоритмы трансляции UCM-спецификаций (Use Case Maps) в раскрашенные сети Петри (CPN). UCM-спецификации позволяют описывать функциональные требования к системе в графической нотации, изображающей сценарии как совокупность причинно-следственных связей между событиями в системе, которые опционально привязаны к ее архитектуре. Рассматриваются все основные стандартные конструкции UCM за исключением сложных конструкций декомпозиции. Трансляция UCM-спецификаций в CPN модель проиллюстрирована примером.

Работа частично поддержана грантом РФФИ № 11-07-90412-Ukr_f_a.

© Институт систем информатики им. А. П. Ершова СО РАН, 2012

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

N.V. Vizovitin, V.A. Nepomniaschy

UCM-SPECIFICATIONS TO COLORED PETRI NETS TRANSLATION ALGORITHMS

Preprint
168

Novosibirsk 2012

Algorithms for translation of UCM (Use Case Maps) specifications to colored Petri nets (CPN) are described. A UCM specification allows for modeling of functional requirements in a concise and efficient manner. The UCM visual scenario notation focuses on the causal flow of behavior optionally superimposed on a structure of components. Translation of all standard UCM constructs is described, with the exception of complex decomposition constructs. Translation of UCM specifications to CPN is illustrated with an example.

© A. P. Ershov Institute of Informatics Systems, 2012

1. ВВЕДЕНИЕ

Широко известна высокая стоимость ошибок на первых этапах разработки программного обеспечения и протоколов, особенно на этапе формализации и анализа требований. Существует немало техник моделирования, позволяющих минимизировать такие ошибки. Одной из нотаций, позволяющих формализовать и анализировать функциональные требования, является графическая нотация Use Case Maps (UCM). UCM-спецификации особенно часто используются для задания требований к телекоммуникационным системам и протоколам, но эта нотация имеет универсальный характер. Например, UCM используется при генерации тестовых сценариев [6; 8].

Диаграммы UCM изображают совокупность сценариев как множество причинно-следственных связей между событиями в системе. При этом события могут быть привязаны к структуре компонентов системы, отражая ее архитектуру. UCM описывает взаимодействие архитектурных сущностей системы, акцентируя внимание на причинно-следственных связях и абстрагируясь от деталей передачи сообщений и обработки данных.

Аппарат анализа и проверки корректности UCM-спецификаций недостаточно развит. Стандарт [2], описывающий UCM, предлагает средство анализа моделей, но оно является достаточно примитивным, и его использование трудоемко.

Предлагаемый подход к проблеме анализа и верификации UCM-спецификаций заключается в их переводе в раскрашенные сети Петри (CPN) [3] и использовании известной системы CPN Tools [10] для анализа раскрашенных сетей Петри. Для их верификации методом проверки моделей разрабатывается новая система, которая использует известную систему SPIN. Раскрашенные сети Петри являются развитым и выразительным формализмом, сочетающим в себе классические сети Петри и функциональный язык.

Цель настоящей работы – описать алгоритм трансляции UCM-спецификаций в раскрашенные сети Петри, включая ограничения алгоритма на исходную модель. В работе рассматриваются только UCM-спецификации с семантикой, описанной в стандарте.

Работа состоит из девяти разделов. В разделе 2 дается обзор языка UCM. Предполагается, что читатель уже знаком с раскрашенными сетями Петри [4]. С 3-го по 8-й разделы описывается алгоритм трансляции

UCM-спецификаций в CPN. В разделе 3 дается общее описание алгоритма. Раздел 4 описывает трансляцию выражений и действий. Раздел 5 описывает трансляцию линейных конструкций UCM, раздел 6 – конструкций альтернативного выбора и параллелизма, раздел 7 – конструкций ожидания. В разделе 8 представлена трансляция конструкций декомпозиции UCM-спецификаций. В разделе 9 обсуждаются перспективы развития алгоритма и проверки корректности UCM-спецификаций. В приложениях показан пример, используемый в данной работе, и приведен результат его трансляции.

2. ОБЗОР ЯЗЫКА UCM

2.1. Обзор графической нотации UCM

UCM является подмножеством языка URN (User Requirements Notation) и частью соответствующего стандарта [2]. UCM – это графическая нотация с достаточно простым синтаксисом. Одна UCM-спецификация или модель может состоять из одной или более диаграмм (map; здесь и далее в скобках будем указывать оригинальные термины, используемые в стандарте, т.к. русскоязычную терминологию, связанную с UCM, вряд ли можно считать устоявшейся), каждая из которых описывает граф элементов, связанных направленными путями. UCM-спецификации используют пути сценариев для изображения причинно-следственных связей между событиями (responsibilities) в системе. Событие в системе – это некоторое наблюдаемое действие, которое может быть выполнено. Событие может быть сопоставлено компоненту (component).

В UCM компонент – достаточно общая и абстрактная сущность, которая может представлять как программные сущности (например, объекты, процессы, базы данных или сервера), так и непрограммные (например, актеров или аппаратное обеспечение). Компоненты задают структуру системы, в то время как пути и другие элементы UCM – ее поведение. Так как компоненты не имеют выделенной семантики и в большинстве случаев, за исключением компонентов с атрибутом protected и компонентов типа Object, не оказывают влияния на семантику других элементов, в данной работе рассматриваются только UCM-модели без привязки к компонентам.

Все конструкции UCM приведены на рис. 1. Диаграммы UCM могут содержать произвольное число путей и компонентов. Пути выражают при-

чинно-следственные связи и могут содержать несколько типов элементов. Путь начинается с элемента типа Start Point и заканчивается элементом типа End Point. Они соответствуют порождающим и результирующим условиям сценария. События, представляемые элементами типа Responsibility и Responsibility Reference, отражают действия или шаги, необходимые для исполнения сценария. Элементы Or-Fork и Or-Join используются для изображения альтернативного выбора. Элементы And-Fork и And-Join позволяют изображать параллелизм. Циклы могут быть смоделированы при помощи элементов Or-Fork и Or-Join. Так как язык UCM не накладывает никаких ограничений на вложенность и взаимное расположение Fork и Join элементов, то их можно свободно комбинировать различными способами, и Join не обязан следовать за Fork того же типа. Элементы типа Waiting Place и Timer обозначают места, в которых исполнение сценария приостанавливается, пока не будет выполнено заданное условие. Отметим, что, несмотря на наличие элемента с именем Timer, в стандартной нотации UCM понятия времени нет. Элементы Empty Point служат исключительно для обеспечения возможности асинхронного соединения путей и собственной семантики не имеют. Как правило, они изображаются либо в виде пустой окружности на пути, либо, что более часто, не изображаются вообще (см., например, рис. 1, Waiting Place). Элементы типа Connect графического представления не имеют и служат для асинхронного или синхронного соединения двух путей (как показано на рис. 1 для элементов Waiting Place и Timer, где элементы Connect и Empty Point не изображены; Waiting Place соединен элементом Connect с Empty Point на искривленном пути; Timer соединен элементом Connect с End Point).

Direction Arrow не является отдельным элементом, а служит для явного отображения направления путей в модели. Этот элемент изображается в виде стрелки на дуге. Использование Direction Arrow предназначено исключительно для прояснения направления дуг в случаях, когда оно не очевидно из контекста. Будем считать, что во всех входных моделях направление всех путей однозначно определено. Это действительно так, поскольку редакторы UCM, например, jUCMNav [9], эту информацию сохраняют.

UCM модели поддерживают декомпозицию при помощи элементов Stub, которые могут содержать дочерние диаграммы, называемые Plug-ins или Plug-in maps. Далее такие диаграммы мы будем называть плагинами. Плагины в UCM – это переиспользуемые единицы поведения и структуры. Связь путей на родительской и дочерних диаграммах определяется через связывания плагина (plug-in bindings), которые определяют, как входные и выходные пути элемента Stub соединены с Start Points и End Points плагина.

Элементы типа Stub могут быть статическими, что означает, что они могут иметь не более одного плагина, или динамическими, которые могут иметь множество плагинов, подмножество которых может быть выбрано на этапе исполнения или симуляции в соответствии с критериями выбора (selection policy). Среди динамических элементов Stub выделяют подтипы: Dynamic Stub, Synchronizing Stub, Blocking Stub. В данной работе рассматривается только Static Stub как наиболее простой и часто употребимый тип.

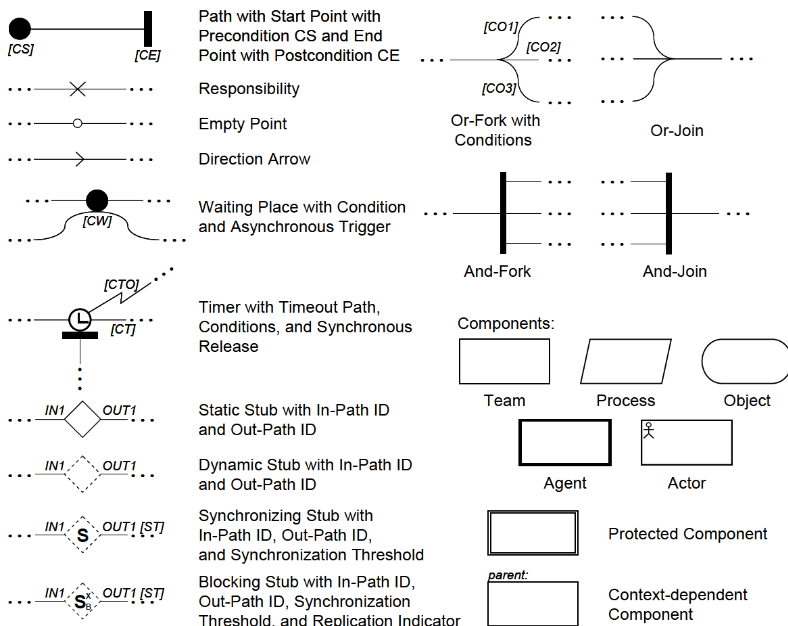

Рис. 1. Синтаксические элементы UCM

На рис. 1 дано графическое представление конструкций UCM. Сверху вниз и слева направо представлены следующие элементы:

- путь с начальной и конечной точкой с предусловием [CS] и постусловием [CE];
- событие – элемент Responsibility или Responsibility Reference;
- Empty Point, изображенный в виде пустого круга;
- Direction Arrow, является вариантом изображения дуги;

- Waiting Place с условием продолжения [CW], асинхронно соединенный при помощи Empty Point (не показан) с искривленным путем;
- Timer с условиями продолжения [CT] и тайм-аута [CTO], с которым синхронно соединен подходящий снизу путь через элемент End Point;
- Static, Dynamic, Synchronizing и Blocking Stubs, их входящие и исходящие пути помечены как IN1 и OUT1;
- Or-Fork с условиями выбора [CO1], [CO2] и [CO3];
- Or-Join;
- And-Fork;
- And-Join;
- разные типы компонентов (Team, Process, Object, Agent, Actor);
- компонент с атрибутом protected;
- контекстно-зависимый компонент, привязка которого к другим элементам зависит от привязки компонентов на родительской диаграмме.

В дополнение к другим ограничениям будем считать, что все элементы в исходной модели уникально идентифицируемы. Это не является существенным ограничением, т.к. все редакторы UCM тем или иным способом идентифицируют отдельные элементы. Также в этой работе не рассматривается семантика атрибутов элементов, связанных с анализом производительности, например, probability или hostDemand. Другие ограничения на отдельные конструкции UCM даны при описании трансляции этих конструкций.

Для удобства изложения введем следующие категории конструкций UCM.

1. Линейные конструкции: Start Point, End Point, Empty Point, Responsibility и Responsibility Reference.
2. Конструкции альтернативного выбора: Or-Fork, Or-Join.
3. Конструкции параллелизма: And-Fork, And-Join.
4. Конструкции ожидания: Waiting Place, Timer.
5. Конструкции декомпозиции: Static Stub и другие типы элементов Stub, Plug-in map.

2.2. Обзор языка данных URN Data Language

UCM позволяет специфицировать действия в рамках событий и условия для различных элементов и исходящих путей. Нотация UCM поддерживает разные уровни формализации моделей (следуя одной из своих целей – пре-

доставлять подходящий уровень формальности модели для заданного этапа в разработке). Один из механизмов, реализующий это – возможность задания действий и выражений либо на естественном языке, либо на языке URN Data Language, являющимся частью стандарта [2]. Иногда некоторые действия и выражения вообще могут быть опущены. Поскольку конечной целью является анализ и проверка корректности UCM модели, будем считать, что все действия и выражения полностью специфицированы на языке URN Data Language.

URN Data Language является простым текстовым языком для описания действий (actions) и выражений (expressions). Действие может изменять состояние некоторых переменных и используется в атрибуте expression элементов Responsibility. Выражения могут быть вычислены и представлены в виде значения соответствующего типа. Наибольший интерес представляют выражения логического типа, используемые в качестве различных условий. URN Data Language основан на подмножестве языка выражений SDL [1] и расширен операторами из популярных языков программирования (например, C и Java). Язык поддерживает 3 основных типа данных: Boolean, Integer и Enumeration. Поддерживаются следующие основные конструкции: логические операторы, операторы сравнения, в том числе равенство и неравенство для всех типов, арифметические операторы, группировка выражений, присвоение, условные операторы, группировка действий.

Переменные в URN являются глобальными и строго типизированными. Переменные, для которых не было определено начальное значение, считаются имеющими специальное значение «undefined». Считаем, что таких переменных в исходной модели нет, и никакая переменная не может принять значение «undefined», т.к. в URN Data Language нет такого литерала. Имя переменной не может заканчиваться на «_pre» – такие имена зарезервированы за константами, хранящими начальные значения соответствующих переменных без этого суффикса.

2.3. Пример UCM-спецификации

В Приложении 1 представлен иллюстративный пример UCM-спецификации, для которого в Приложении 2 дан результат трансляции в CPN. Он приведен в качестве иллюстрации трансляции всех рассматриваемых в данной работе конструкций.

3. ТРАНСЛЯЦИЯ UCM-СПЕЦИФИКАЦИЙ. ВЕРХНИЙ УРОВЕНЬ

Введем следующие обозначения для основных операций и выражений:

s1 + s2 – конкатенация строк s1 и s2.

"$var" (внутри строки) – подстановка значения var в строку.

x := a – присваивание переменной x значения a.

e[x/t] – e, в котором x заменено на t.

∀ x | Cond { Expr } – для любого x, такого, что Cond, выполнить Expr.

Func(arg1 | Cond1, arg2) → Result { Body } – определение реализации функции Func с телом Body, возвращающей результат в переменной Result, и применимой для любого второго аргумента arg2 и первого аргумента arg1, удовлетворяющего условию Cond1. Переменной Result должно быть присвоено некоторое значение в теле функции.

Однострочные комментарии в псевдокоде начинаются с символов «//».

3.1. Представление входных данных

Будем рассматривать алгоритм трансляции как перевод одного ориентированного графа в другой. Первый граф будет представлять исходную модель UCM, а второй – результирующую раскрашенную сеть Петри. Поскольку и исходная, и целевая нотации описывают иерархические структуры, то в общем случае необходимо рассматривать ориентированные иерархические графы, в которых отдельным вершинам могут соответствовать свои графы. Однако в случае, если исходная UCM модель не содержит элементов типа Stub, можно рассматривать неиерархические графы. При этом все построения естественным образом расширяются на случай иерархических графов. Далее, если не оговорено иначе, будет рассматриваться неиерархический случай, т. е. исходные модели, не содержащие элементов Stub.

Входными данными будем считать ориентированный граф G = < V, E >, вершинами V которого являются элементы UCM, являющиеся наследниками PathNode в метамодели нотации (см. стандарт [2]), а дугами E – элементы NodeConnection, т. е. отрезки пути между другими элементами диаграммы. Определим на элементах входных данных следующие функции. Ниже v, u ∈ V и e ∈ E.

uid(v) – уникальное имя элемента (строка). Если элемент был в исходном документе, то это, как правило, его имя. В этом случае существование такого идентификатора обосновывается требованиями к входным данным. Иначе это некоторый уникальный идентификатор, что обеспечивается реализацией алгоритма.

in(v) – множество смежных с v вершин, инцидентных входящим дугам.
out(v) – множество смежных с v вершин, инцидентных исходящим дугам.
type(v) – тип элемента UCM (например, OrFork, Timer).
arc(v, u) = e ∈ E | e = (v, u).
action(v) – Responsibility expression на URN Data Language, если type(v) ∈ { Responsibility, RespRef }.
expr(v) – выражение на URN Data Language, привязанное к элементу v, type(v) ∈ { StartPoint, EndPoint, WaitingPlace }.
expr(e) – выражение на URN Data Language, привязанное к исходящей дуге e = (v, u) | type(v) ∈ { Timer, OrFork }.
isSimpleElem(v) = v ∈ { Responsibility, RespRef, StartPoint, EndPoint, EmptyPoint, OrFork, OrJoin, AndFork, AndJoin, WaitingPlace, Timer }.
isStub(v) = v ∈ { StaticStub }.
isPrimaryElem(v) = isSimpleElem(v) ∨ isStub(v).
hasConnect(v) = ∃ u ∈ in(v) ∪ out(v) | type(u) = Connect.

Далее будем рассматривать только следующие элементы: ∀ v ∈ V isPrimaryElem(v) ∨ type(v) = Connect. Другими словами, будем рассматривать все элементы, кроме нестатических Stubs и компонентов.

3.2. Представление выходных данных

Выходными данными алгоритма является раскрашенная сеть Петри [3], соответствующая исходной UCM модели. Представим ее в виде двудольного ориентированного графа H = < T, P, A, D >, где T – множество переходов, P – множество мест, A – множество дуг, D – набор определений (в том числе цветов). Ниже t ∈ T, p ∈ P, a ∈ A.

Определим конструкторы элементов графа.

t = **trans**(name, guard=empty) – новый переход с уникальным именем name и охранным выражением (guard condition) guard. По умолчанию охранное выражение отсутствует (empty).

p = **place**(name, colset=UNIT, init=empty) – новое место с уникальным именем name, цветом или типом фишек colset и начальной разметкой init. По умолчанию начальная разметка пуста и имеет значение empty.

a = **arc**(t, p, expr=()) или a = **arc**(p, t, expr=()) – новая дуга от перехода к месту или от места к переходу с выражением expr. Значение выражения по умолчанию – (), единственное допустимое значение цвета UNIT.

Заметим, что если a = arc(t, p, expr) или a = arc(p, t, expr) и p = place(name, colset, init), то color(expr) = colset. Здесь **color(v)** – цвет значения выражения v.

Отметим следующие свойства выходных данных. Полученные CPN не являются временными (timed), т.к. в рассматриваемой семантике UCM не используется понятие времени. Результирующая CPN модель является иерархической тогда и только тогда, когда исходная UCM модель также была иерархической, т. е. имела конструкции типа Stub.

3.3. Описание алгоритма

Алгоритм трансляции UCM-спецификаций в раскрашенные сети Петри носит локальный характер в том смысле, что для трансляции фрагмента исходной модели не требуется рассматривать всю модель. Алгоритм может быть разбит на 4 основных шага:

1) Определение общих объявлений для CPN – цветов, констант и переменных.
2) Разбиение некоторых дуг исходного графа UCM на две вставкой новых элементов типа FakePathNode. Этот шаг можно опустить и рассматривать вместо этих новых элементов исходные дуги, но он позволяет изложить алгоритм в более удобной и простой форме.
3) Трансляция каждого элемента полученной модели, кроме элементов типа Connect и FakePathNode, отдельно от других вместе со своей непосредственной окрестностью.
4) Объединение получившихся на предыдущем шаге графов и новых объявлений.

В целом алгоритм может быть описан следующим образом на псевдокоде:

```
Translate(G) → H
{
  // шаг 1 – определение общих объявлений
  Decls = DefineColors(G) ∪ DefineConstants(G) ∪ DefineVars(G)

  // шаг 2 – введение элементов типа FakePathNode
  G1 = AddFakeNodes(G)

  // шаги 3 и 4 – трансляция элементов по отдельности
  // и объединение получившихся графов и новых объявлений
  H := < ∅, ∅, ∅, Decls >
```

```
  ∀ v ∈ V(G1) | isPrimaryElem(v)
  {
      H := H ∪ TranslateElem(v, Decls)
  }
}
```

Приведенный алгоритм применим для исходных моделей без элементов типа Stub либо только для верхнего уровня модели. Его расширение на случай наличия элементов типа Stub и дочерних диаграмм приведено в соответствующем разделе.

Для функционирования раскрашенной сети Петри необходимо ввести следующие служебные цвета:

1) colset UNIT = unit; – стандартный «базовый» цвет с единственным допустимым значением «()». Фишки этого цвета используются как транспорт сигналов из исходной модели.
2) colset SATISFIED = bool; – булевый цвет, используется для различного рода флагов.
3) colset ACTIVATION = subset INT by (fn x => x >= 0); – цвет неотрицательных целых чисел. Используется для учета накапливаемых сигналов.

Кроме этого понадобятся цвета, соответствующие типам данных в исходной модели. Базовых типов всего три – Boolean, Integer и Enumeration. Для каждого используемого Enumeration необходимо определить свой цвет как enumeration color set (используя ключевое слово with). В результате типы преобразуются в цвета следующим образом:

1) тип Boolean – colset BOOL = bool;
2) тип Integer – colset INT = int;
3) тип Enumeration с именем name и элементами перечисления id0, id1, …, idn – colset name = with id0 | id1 | … | idn;. Например, colset Day = with Mon | Tue | Wed | Thu | Fri | Sat | Sun.

Описанное полностью определяет функцию DefineColors(G) → C. Отметим, что количество цветов не превышает 5+N, где N – количество различных Enumeration типов, которое, в свою очередь, не больше количества переменных.

Константы в UCM определяются как переменные инициализации или как так называемые pre variables и отличаются от других переменных наличием суффикса "_pre" в имени. Такие константы преобразуются в константы CPN, определяемые как val name_pre = value;. Заметим, что неиспользуемые в исходной модели константы можно не транслировать. Также для

упрощения CPN можно транслировать отдельные переменные, которым никогда не присваивается другое значение, как константы. Этим полностью определяется функция DefineConstants(G) → C.

Определение переменной выглядит следующим образом: var var_name : COLOR;. Здесь var_name – имя переменной, COLOR – ее цвет, color(var_name) = COLOR. Также на дугах, инцидентных местам цвета ACTIVATION, будет использоваться переменная var act : ACTIVATION. Это полностью определяет функцию DefineVars(G) → C.

Для удобства последующего изложения, расширим абстрактную грамматику UCM новым элементом FakePathNode, являющимся наследником PathNode. Графически изобразим эти новые элементы как серые круги на дугах. Вершины типа FakePathNode не несут никакой семантической нагрузки, а служат лишь как удобное средство определения окрестности других элементов. В дальнейшем эти вершины будут преобразованы в места CPN с цветом UNIT. Ниже приведен псевдокод алгоритма добавления вершин нового типа.

```
AddFakeNodes(G) → F
{
  V = V(G)
  E = E(G)
  // новый граф содержит все вершины исходного графа
  F := < V, Ø >
  // для каждой пары рассматриваемых элементов, кроме Connect
  ∀ v, u ∈ V | e = (v, u) ∈ E, isPrimaryElem(v), isPrimaryElem(u)
  {
     // дуга между ними разбивается новой вершиной w
     w := FakePathNode()
     // и в новый граф добавляются вершина w и две производные
     // дуги, при этом только дуга, входящая в вершину w, сохраняет
     // атрибуты оригинальной дуги, если они присутствовали
     F := F ∪ < w, {e[u/w], (w, u)} >
  }
  // дуги между всеми остальными парами элементов добавляются
  // без изменений
  ∀ v, u ∈ V | e = (v, u) ∈ E, type(v) = Connect ∨ type(u) = Connect
  {
     F := F ∪ < Ø, {e} >
  }
}
```

В этом коде FakePathNode() – конструктор нового элемента типа FakePathNode. Отметим, что конструируемый элемент, как и все другие, является уникально идентифицируемым. Как видно из этого фрагмента кода, исходный граф модифицируется следующим образом. Каждая дуга между рассматриваемыми вершинами, ни одна из которых не имеет тип Connect, разбивается на две новой вершиной типа FakePathNode. При этом свойства исходящих дуг сохраняются (например, выражения, сопоставленные им в исходной модели). Направление дуг также сохраняется. Все остальные дуги исходного графа не модифицируются. Пример работы этой функции приведен на рис. 2.

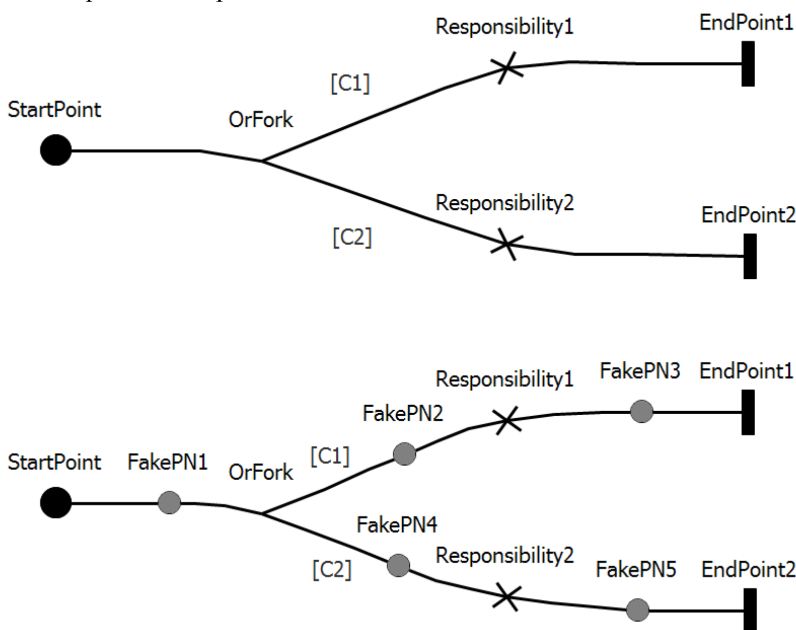

Рис. 2. Разбиение дуг исходного графа элементами FakePathNode

Таким образом, в новом графе F любая вершина $v \in V(F) \mid isPrimaryElem(v)$ может иметь в качестве смежных вершин лишь вершины типа Connect и FakePathNode. Поэтому далее в качестве окрестности транслируемой вершины будем рассматривать инцидентные ей дуги и смежные с ней вершины. Сама вершина при этом будет транслирована в переход,

смежные вершины типа Connect – в места типа ACTIVATION, а смежные вершины типа FakePathNode – в места типа UNIT. Инцидентные транслируемой вершине дуги будут преобразованы в дуги CPN модели. Также могут появиться другие вспомогательные места и дуги, а также места и дуги, соответствующие используемым в транслируемой окрестности переменным.

После трансляции отдельных фрагментов расширенной исходной модели необходимо объединить полученные графы и новые определения. Определения объединяются естественным образом – объединение включает в себя все определения из объединяемых частей без повторений. При этом алгоритм трансляции отдельных элементов должен гарантировать, что не возникнет конфликтующих определений. Полученные после трансляции отдельных элементов графы также объединяются естественным образом – как множества вершин и дуг. При этом за счет того, что все элементы уникально идентифицируемы, и в силу выбора окрестности элемента для трансляции объединение фактически всегда происходит по местам, соответствующим элементам границы транслируемого фрагмента, а также местам переменных. Таким образом, объединение заключается в «склейке» полученных фрагментов по соответствующим местам типа UNIT, ACTIVATION и местам переменных.

4. ТРАНСЛЯЦИЯ ВЫРАЖЕНИЙ И ДЕЙСТВИЙ

В общем случае для трансляции выражения или действия из URN Data Language в CPN и CPN ML необходимо построить необходимые места используемых переменных, инцидентные им дуги, а также транслировать само выражение в CPN ML. При трансляции выражений места переменных и инцидентные им дуги строятся тривиальным образом, поскольку выражения не изменяют значения переменных. Достаточно построить места для всех используемых переменных, а пометки на дугах должны соответствовать имени переменной, месту которой они инцидентны. Трансляция выражений использует упомянутые выше имена переменных и подробно описана в соответствующем подразделе.

Основная сложность трансляции действий заключается в подготовке группы выражений для дуг, входящих в места переменных, которые отражают, как соответствующие переменные должны изменить свое значение после исполнения действия. Переменные, встречающиеся только в правой части присваиваний в заданном действии, также транслируются тривиаль-

ным образом подобно переменным, используемым в выражении. Трансляция действий подробно рассматривается в соответствующем подразделе.

4.1. Трансляция выражения или действия во фрагмент раскрашенной сети Петри

Определим несколько функций для трансляции выражений (expr) и действий (action) на URN Data Language.

rvalues(action) или **rvalues(expr)** – список кортежей <name, type, value> из имени, типа и начального значения соответственно, определяющих переменные, присутствующие в выражении или действии справа от знака присваивания. Для выражений это, фактически, все встречающиеся переменные.

lvalues(action) – список кортежей <name, type, value>, определяющих переменные, присутствующие в действии слева от знака присваивания.

lvalues(expr) = ∅ – для выражений значение lvalues(expr) пусто, т.к. в них не встречаются присваивания.

Типы URN Data Language преобразуются в цвета CPN с помощью функции **typeToCol(type)** следующим образом:

typeToCol(Boolean) = BOOL
typeToCol(Integer) = INT
typeToCol(enumName) = enumName – перечисления преобразуются в соответствующие им перечисления, порожденные в CPN ML на первом шаге алгоритма.

ExprToML(expr) → ML – функция, которая транслирует выражение на URN Data Language в выражение на языке CPN ML. Результат использует аргументы, соответствующие пометкам на дугах, генерируемых в функции TranslateExpr(). Реализация данной функции описана ниже в подразделе о трансляции выражений.

ActionToML(action, var) → ML – функция, которая транслирует действие на URN Data Language в выражение на языке CPN ML, значение которого соответствует значению переменной var после выполнения исходного действия. Реализация данной функции описана ниже в подразделе о трансляции действий.

Приведем функцию трансляции выражения или действия expr для окрестности элемента, транслированного в переход t, во фрагмент CPN H. Эта функция строит фрагмент сети с необходимыми местами переменных, а также дугами между ними и переходом t.

```
TranslateExpr(expr, t, Decls) → H
{
  R = rvalues(expr)
  L = lvalues(expr)
  if L = Ø {      // если нет присваиваний, то есть expr – выражение
     H :=   <  // построим фрагмент графа, содержащий переход t
              { t },
              // места для каждой встречающейся переменной
              { place(name, typeToCol(type), value)
                  ∀ <name, type, value> ∈ R },
              // двунаправленные дуги, помеченные именем
              // переменной, между каждым местом переменной
              // и переходом t
              { arc(t, p, name), arc(p, t, name)
                  ∀ name, p | <name, type, value> ∈ R,
                  p = place(name, typeToCol(type), value) },
              Decls
           >
  } else {        // если expr – действие
     // построим фрагмент графа, содержащий места переменных
     P := { place(name, typeToCol(type), value)
              ∀ <name, type, value> ∈ R ∪ L }
     // входящие в переход t дуги из мест переменных для «чтения»,
     // помеченные именами соответствующих переменных
     RA := { arc(p, t, name)
              ∀ name, p | <name, type, value> ∈ R ∪ L,
              p = place(name, typeToCol(type), value) }
     // исходящие из перехода t дуги в места переменных,
     // помеченные результатами трансляции действия expr для
     // данных переменных с именем name
     WA := { arc(t, p, ActionToML(expr, name))
              ∀ name, p | <name, type, value> ∈ R ∪ L,
              p = place(name, typeToCol(type), value) }
     H := < { t }, P, RA ∪ WA, Decls >
  }
}
```

Заметим, что, если имя переменной name, то пометки на дугах, исходящих из места этой переменной, также будут name. Пометки на входящих дугах либо также name, в случае если данную переменную только читают, либо результат применения функции ActionToML(action, name), в случае

если в транслируемом действии данной переменной присваивается новое значение.

4.2. Трансляция выражений

В данном подразделе рассмотрим реализацию функции ExprToML(expr) → ML, которая осуществляет трансляцию выражений URN Data Language в выражения CPN ML. Грамматика выражений URN Data Language описана в стандарте [2], раздел 9.3. Выражения не могут включать сложных конструкций, а лишь состоят из переменных, констант, литералов, операторов и скобок.

Поскольку синтаксисы выражений исходного и целевого языка очень похожи, для трансляции мы можем ограничиться простой заменой операторов. Интерпретация литералов в этих языках совпадает, а переменные, константы и значения перечислений транслируются без изменений в силу свойств алгоритма трансляции определений. Таблица 1 определяет правила замены операторов в выражении. Все допустимые в исходном выражении операторы перечислены во втором и третьем столбцах таблицы. Эти операторы необходимо заменить на соответствующие им из четвертого столбца.

Таблица 1. Замена операторов при трансляции выражений

Оператор	Синтаксис в URN Data Language		Синтаксис в CPN ML
	Синтаксис SDL	Альтернативный синтаксис	
Равно	=	==	=
Не равно	/=	!=	<>
Больше чем	>	>	>
Меньше чем	<	<	<
Больше или равно	>=	>=	>=
Меньше или равно	<=	<=	<=
Сложение	+	+	+
Вычитание	–	–	–
Умножение	*	*	*
Деление (целочисленное)	/	/	div
Остаток от деления	mod	%	mod
Унарный минус	–	–	~
Отрицание	not	!	not

<table>
<tr><td>Конъюнкция</td><td>and</td><td>&&</td><td>andalso</td></tr>
<tr><td>Дизъюнкция</td><td>or</td><td>||</td><td>orelse</td></tr>
<tr><td>Исключающее ИЛИ</td><td>xor</td><td>^</td><td>xor</td></tr>
<tr><td>Импликация</td><td>=></td><td>=></td><td>implies</td></tr>
</table>

В CPN ML нет операторов исключающего ИЛИ и импликации, поэтому необходимо их определить. Сделаем это следующим образом:

```
fun xor(a, b) = a <> b;                fun implies(a, b) = not a orelse b;
infix xor;                             infix implies;
```

Таким образом, функция ExprToML(expr) состоит из следующих шагов: разобрать выражение expr на лексемы, заменить все операторы в соответствии с приведенной выше таблицей, интерпретировать полученный поток лексем как требуемое выражение CPN ML.

4.3. Трансляция действий

В этом подразделе мы рассмотрим реализацию функции ActionToML(action, var) → ML, осуществляющей трансляцию действий, выраженных на URN Data Language, в выражения CPN ML. Результат применения этой функции отражает значение переменной var после выполнения действия action. Грамматика выражений URN Data Language описана в стандарте [2], раздел 9.4. Действие может представлять собой простое присваивание, составное действие или условное выражение. Тип любого действия – Void, то есть действия не могут быть использованы там, где требуется значение, в частности, действия используются только в элементах типа Responsibility. Ниже приведен фрагмент грамматики действий.

```
<action>           ::= <statement>*
<statement>        ::= <assignment> | <compound statement>
                     | <if statement>
<assignment>       ::= <variable name> <assignment operator>
                     <expression> <statement terminator>
<compound statement> ::= <left curly bracket> <statement>*
                             <right curly bracket>
<if statement>     ::= <if> <left parenthesis> <expression>
                     <right parenthesis> <statement>
                     [<else> <statement>]
```

Определим функцию ActionToML() как ActionToML(stmts, var) = StmtsToML(stmts, var). Функция StmtsToML(stmts) преобразует упорядоченный список предложений stmts исходного языка в выражение CPN ML, соответствующее значению переменной var после исполнения этого списка предложений.

```
StmtsToML(stmts, var) → ML    // stmts – список предложений
{
  if stmts = ∅ {
     ML := “”
  } else {
     ML := “let “
     ∀ statement ∈ stmts   // итерация в соответствии с порядком
     {
        ML := ML + StmtToLet(statement) + “ “
     }
     ML := ML + “in $var end”
  }
}
```

Функция StmtToLet(statement) преобразует предложение statement исходного языка в одно или более выражений определения констант с именами присваиваемых в этом предложении переменных. Полученные выражения используются в конструкции let-in-end (см. выше). Ниже value(symbol, order=1) → String – специальная функция, возвращающая по символу symbol и его порядковому номеру order из правой части правила в грамматике его значение в данном выражении. Например, если StmtToLet() вызван с аргументом “x := y + 1”, то value(<variable name>) = “x”, а value(<expression>) = “y + 1”.

```
StmtToLet(<variable name> <assignment operator>
            <expression> <statement terminator>) → ML
{
  var = value(<variable name>)
  expr = ExprToML( value(<expression>) )
  // присваивание преобразуется в определение новой
  // локальной константы со значением присваиваемого выражения
  ML := “val $var = $expr”
}
```

```
StmtToLet(<left curly bracket> <statement>*
            <right curly bracket>) → ML
{
  stmts = value(<statement>*)
  ML := “”
  // группа предложений преобразуется в группу
  // определений констант
  ∀ statement ∈ stmts
  {
     ML := ML + StmtToLet(statement) + “ “
  }
}

StmtToLet(<if> <left parenthesis> <expression>
            <right parenthesis> <statement>) → ML
{
  expr = ExprToML( value(<expression>) )
  stmt = value(<statement>)
  L = lvalues(stmt)
  ML := “”
  // условные операторы преобразуются в группу определений
  // констант, каждая из которых равна результату аналогичного
  // оператора в CPN ML; определяется константа для каждой
  // переменной, присваиваемой в исходном предложении
  ∀ var ∈ L
  {
     thenML := StmtsToML(stmt, var)
     ML := ML + “val $var = if $expr then $thenML else $var “
  }
}

StmtToLet(<if> <left parenthesis> <expression>
            <right parenthesis> <statement>
            <else> <statement>) → ML
{
  expr = ExprToML( value(<expression>) )
  thenStmt = value(<statement>, 1)
  elseStmt = value(<statement>, 2)
  L = lvalues(thenStmt) ∪ lvalues(elseStmt)
  ML := “”
  // случай, когда есть else-ветвь в if, аналогичен предыдущему
```

```
  ∀ var ∈ L
  {
      thenML := StmtsToML(thenStmt, var)
      elseML := StmtsToML(elseStmt, var)
      ML := ML + “val $var = if $expr then $thenML else $elseML “
  }
}
```

Таким образом, любое действие в CPN ML представляется как группа выражений по числу присваиваемых переменных, каждое из которых с помощью конструкций let-in-end и if-then-else отражает изменение отдельной переменной при исполнении данного действия. Тем самым функция ActionToML(action, var) → ML описана.

4.4. Примеры

Рассмотрим примеры трансляции выражений (см. также Приложение 2). Функция ExprToML() переводит выражение “! y” на дуге OrFork-F11 в выражение CPN ML “not y”, а выражение “z != Mid” из действия элемента Resp3 в выражение CPN ML “z <> Mid”.

Рассмотрим примеры трансляции действий. Сначала рассмотрим пример достаточно простого действия, сопоставленного элементу Resp2: “x = 10; z = Mid;”. Применение функции TranslateExpr(“x = 10; z = Mid;”, Resp2T, Decls) породит следующий фрагмент CPN, приведенный на рис. 3.

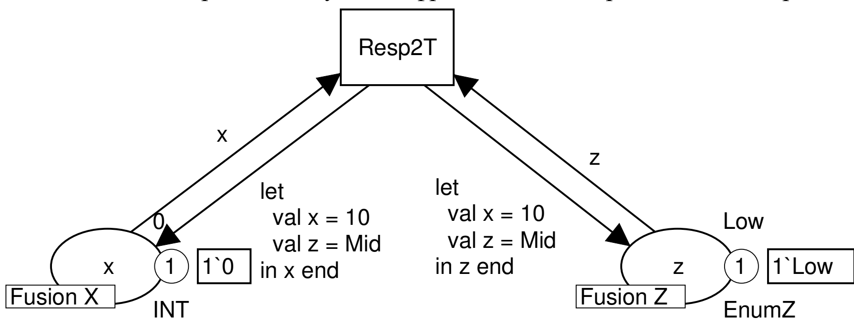

Рис. 3. Результат применения функции TranslareExpr()

Заметим, что сложные выражения на дугах получены в результате применения функции ActionToML(). Их, очевидно, можно упростить, однако мы не будем рассматривать эту техническую задачу в данной работе.

Рассмотрим трансляцию действия элемента Resp3. Опишем здесь лишь результаты работы вызовов ActionToML(expr(Resp3), x) и ActionToML(expr(Resp3), y), обернутые в функции Resp3X() и Resp3Y() соответственно (см. также Приложение 2).

```
fun Resp3X(x, y, z) =
   let
       // определения y и x – результат StmtToLet(action), где
       // action – это исходное действие (весь условный оператор)
       val y =
           if x >= 5 then
               let
                   val y = true
                   val x = 5
               in y end
           else
               // результат StmtsToLet("y=false; if (z!=Mid) x = 2*x + 1;")
               let
                   val y = false
                   // результат StmtToLet("if (z != Mid) x = 2*x + 1")
                   val x = if z <> Mid then let x = 2*x + 1 in x end
               in y end
       val x =
           if x >= 5 then
               let
                   val y = true
                   val x = 5
               in x end
           else
               let
                   val y = false
                   val x = if z <> Mid then let x = 2*x + 1 in x end
               in x end
   in x end;
```

Реализация Resp3Y(x, y, z) полностью аналогична за исключением того, что на верхнем уровне используется конструкция let … in y end;. Очевидно, эти функции также можно значительно упростить.

5. ТРАНСЛЯЦИЯ ЛИНЕЙНЫХ КОНСТРУКЦИЙ

В данном разделе мы рассмотрим трансляцию линейных конструкций: Start Point, End Point, Empty Point и Responsibility. Но поскольку элементы многих типов могут быть смежны с элементами типа Connect, сначала рассмотрим общий подход к трансляции этих элементов.

5.1. Connect

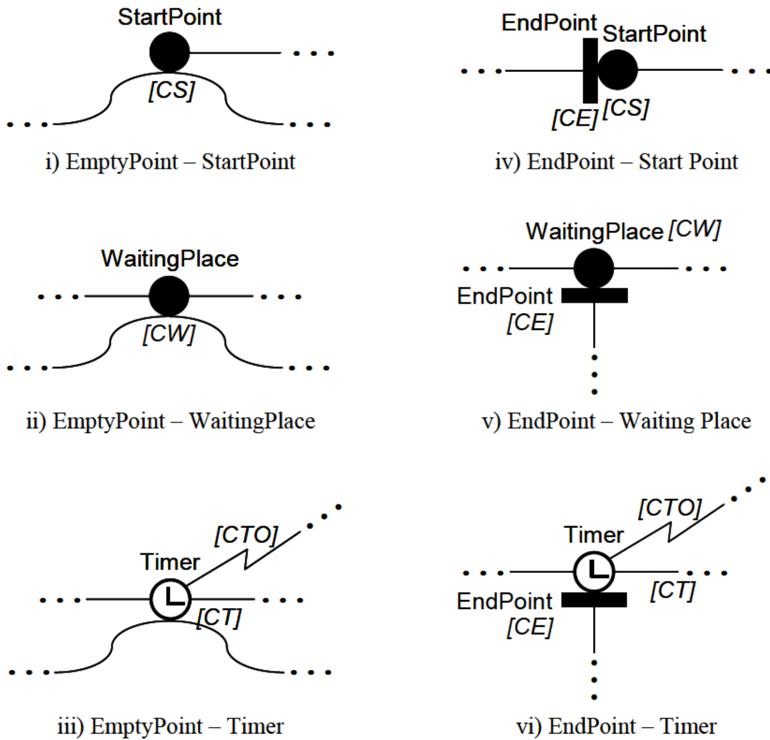

Рис. 4. Варианты использования элементов Connect и Empty Point

Connect – это элемент UCM, который позволяет соединить ровно два пути друг с другом либо синхронно (т. е. последовательно, соединяя End Point и другой путь), либо асинхронно (соединяя Empty Point с другим пу-

тем). Connect не имеет конкретного синтаксиса и не отображается напрямую. Вместо этого Connect влияет на изображение других элементов, которые он соединяет (см. рис. 4). Если для соединения путей используется Empty Point, то ее символ также не изображается, а вместо этого путь, на котором она лежит, изображается искривленным.

Элементы типа Connect всегда транслируются вместе со смежными с ними элементами как часть их окрестности. Каждому элементу Connect соответствует место типа ACTIVATION с начальной разметкой 1`0. Между смежными друг с другом местом типа ACTIVATION и переходом всегда есть две дуги, одна из них с выражением “act”, а другая с выражением, уменьшающим или увеличивающим значение фишки на месте. Используется ли уменьшение или увеличение значения, зависит от направления дуги в исходной модели.

Ниже приведена функция, порождающая подмножество модели CPN с местом цвета ACTIVATION, смежным с переходом t, соответствующим элементу v в UCM модели, с направлением дуги от элемента Connect direction. Также пользователь этой функции может переопределить выражение модификации фишки на месте ACTIVATION посредством задания параметра actExpr. Если заданный элемент не смежен с Connect, то эта функция не порождает новых элементов.

```
AddConnectTrigger(v | v ∈ V, t, Decls, direction | direction ∈ {IN, OUT},
actExpr = Ø) → H
{
  if hasConnect(v) {
      if direction = IN {
          // смежный с v элемент типа Connect
          connect = u ∈ in(v) | type(u) = Connect
          if actExpr = Ø {
              // ACTIVATION не допускает отрицательных значений
              actExpr = “if act > 0 then act-1 else 0”
          }
      } else {         // direction = OUT
          connect = u ∈ out(v) | type(u) = Connect
          actExpr = “act+1”
      }

      // новое место типа ACTIVATION соединяется с переходом для
      // элемента v двумя дугами с пометками act и actExpr
      trigger = place( uid(connect), ACTIVATION, 0 )
```

```
      H :=   <  {t}, {trigger},
                { arc(trigger, t, “act”), arc(t, trigger, actExpr) },
                Decls  >
   } else {
      H := Ø
   }
}
```

Место trigger, создаваемое в этой функции, всегда имеет ровно одну фишку цвета ACTIVATION, значение которой далее будем называть активацией. Активация равна количеству прошедших через Connect и еще не обработанных сигналов. Под проверкой активации далее будем иметь в виду проверку того, что активация положительна. Заметим, что активация всегда неотрицательна.

5.2. Start Point

Start Point обозначает начало сценария. Предусловие Start Point выражает условия, для которых определен сценарий. Если предусловие истинно – сценарий может начаться с данного элемента Start Point. Иначе сценарий не может быть начат. Элемент типа Start Point может иметь связь с Connect. Элементы Start Point также играют роль в иерархическом структурировании UCM спецификаций. Этот случай рассмотрен в разделе «Трансляция конструкций декомпозиции».

При трансляции Start Point (аргумент v ниже) преобразуется в переход, имеющий в качестве guard condition свое предусловие. В guard также добавляется проверка активации, если Start Point соединен с Connect. Смежный элемент (fakeOut) типа FakePathNode транслируется в место типа UNIT с пустой начальной разметкой (out). Для того чтобы сценарий мог начаться, добавляется место start типа UNIT с одной фишкой в начальной разметке. Также необходимо добавить связь с местами всех переменных, используемых в предусловии expr(v).

```
TranslateElem(v | v ∈ V & type(v) = StartPoint, Decls) → H
{
   start = place( uid(v), UNIT, () )
   fakeOut = u | u ∈ out(v), type(u) = FakePathNode
   out = place( uid(fakeOut) )
```

```
   if hasConnect(v) {
       t = trans( uid(v)+"T", "[" + ExprToML(expr(v)) + ", act > 0]" )
   }  else {
       t = trans( uid(v)+"T", "[" + ExprToML(expr(v)) + "]" )
   }
   H := < {t}, {start, out}, { arc(start, t), arc(t, out) }, Decls >
   H := H     ∪ AddConnectTrigger(v, t, Decls, IN, "act-1")
              ∪ TranslateExpr(expr(v), t, Decls)
}
```

Пример применения функции TranslateElem() к элементу TrigStart приведен на рис. 5. Здесь и далее в качестве примера трансляции отдельных элементов приведена трансляция элементов примера, приведенного в Приложении 1.

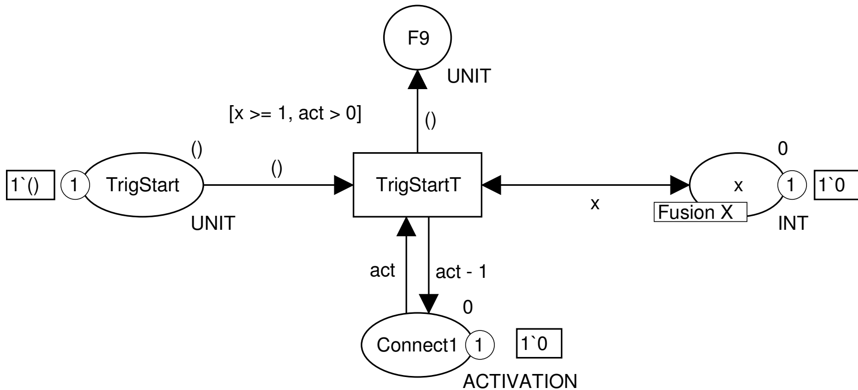

Рис. 5. Трансляция StartPoint

В этом примере место F9 соответствует добавленному элементу FakePathNode (см. Приложение 2). Переход TrigStartT смежен с местом типа ACTIVATION, поскольку в исходной модели стартовая точка TrigStart смежна с элементом типа Connect. Место x является местом, соответствующим переменной x, используемой в предусловии, отраженном в охранном выражении, которое также осуществляет проверку активации. Место x и смежные с ним дуги были порождены функцией TranslateExpr(), а место Connect1 – функцией AddConnectTrigger().

5.3. End Point

End Point – элемент UCM, обозначающий конец локального сценария, для которого может быть определено постусловие. При прохождении End Point осуществляется проверка постусловия. Если оно истинно, считается, что сценарий успешно завершился.

При трансляции элементов типа End Point вводится дополнительное служебное место endPostCond типа SATISFIED с пустой начальной разметкой. Наличие фишек на этом месте является признаком завершения сценария, а их цвет отражает успешность (true) или неуспешность (false) прохождения сценария. В случае если постусловие (expr(v)) отсутствует, считается, что сценарий всегда завершается успешно. Заметим, что неуспешное завершение сценария в общем случае не останавливает другие сценарии.

```
TranslateElem(v | v ∈ V & type(v) = EndPoint, Decls) → H
{
  endPostCond = place( uid(v) + "_PC", SATISFIED, empty )
  fakeIn = u | u ∈ in(v), type(u) = FakePathNode
  in = place( uid(fakeIn) )

  if expr(v) = Ø {
     postCond = true
  } else {
     postCond = ExprToML(expr(v))
  }

  t = trans( uid(v)+"T" )
  H :=   <  {t}, {in, endPostCond},
            { arc(in, t), arc(t, endPostCond, postCond) },
            Decls  >
  H := H    ∪ AddConnectTrigger(v, t, Decls, OUT)
            ∪ TranslateExpr(expr(v), t, Decls)
}
```

На рис. 6 и рис. 7 представлен результат трансляции элементов TrigEnd и End1. TrigEnd смежен с элементом Connect2, который смежен с элементом Timer, последний в данном подразделе не рассматривается. End1 имеет постусловие [v > 3]. Места F14 и F6 также получены при трансляции добавленных элементов типа FakePathNode.

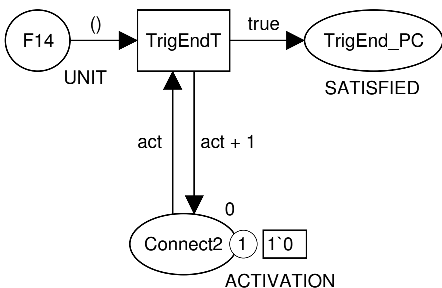

Рис. 6. Трансляция EndPoint, смежного с Connect

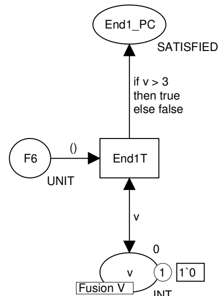

Рис. 7. Трансляция EndPoint с постусловием

5.4. Empty Point

Как уже упоминалось выше, Empty Point, как правило, явно не отображается на диаграммах UCM, но используется для поддержки асинхронного соединения двух путей, см. рис. 4, фрагменты i-iii. Собственной семантики этот элемент не имеет.

При трансляции основная функция элемента Empty Point заключается во введении дополнительного места цвета ACTIVATION.

```
TranslateElem(v | v ∈ V & type(v) = EmptyPoint, Decls) → H
{
   fakeIn = u | u ∈ in(v), type(u) = FakePathNode
   fakeOut = u | u ∈ out(v), type(u) = FakePathNode
   in = place( uid(fakeIn) )
   out = place( uid(fakeOut) )

   t = trans( uid(v)+"T" )
   H := < {t}, {in, out}, { arc(in, t), arc(t, out) }, Decls >
   H := H ∪ AddConnectTrigger(v, t, Decls, OUT)
}
```

На рис. 8 приведена трансляция элемента EmptyPoint, находящегося на отрезке пути между элементами Start и Resp1 или элементами F1 и F2 после добавления дополнительных элементов типа FakePathNode. Транслируемый элемент служит для асинхронного соединения с элементом TrigStart через Connect1.

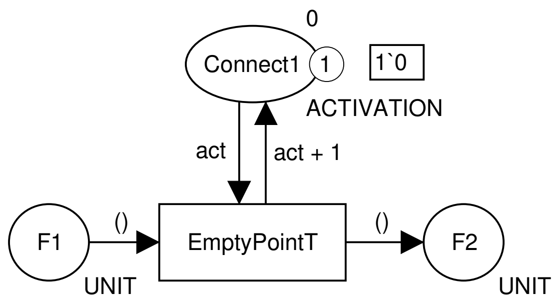

Рис. 8. Трансляция EmptyPoint

5.5. Responsibility

Стандарт выделяет элементы Responsibility (также называемый responsibility definition) и RespRef. Первый задает переиспользуемое определение действия в сценарии, другими словами шаг сценария. На диаграммах UCM на такие определения ссылаются элементы RespRef. Атрибут `expression` элемента Responsibility задает действие формальным образом на URN Data Language.

При трансляции не будем различать элементы Responsibility и RespRef и будем рассматривать последние как те элементы, на которые они указывают. Также не рассматривается семантика атрибута repetitionCount, обозначающего количество повторений действия, которое всегда будем считать равным 1, и hostDemand у элементов типа RespRef.

Рассматриваемые элементы транслируются в переход и дуги к местам используемых в действии переменных. Выражение на дуге соответствуют переведенному в CPN ML действию для данной переменной.

```
TranslateElem(v | v ∈ V & type(v) ∈ {Responsibility, RespRef},
Decls) → H
{
  fakeIn = u | u ∈ in(v), type(u) = FakePathNode
  fakeOut = u | u ∈ out(v), type(u) = FakePathNode
  in = place( uid(fakeIn) )
  out = place( uid(fakeOut) )

  t = trans( uid(v)+”T” )
  H := < {t}, {in, out}, { arc(in, t), arc(t, out) }, Decls >
```

```
    H := H ∪ TranslateExpr(expr(v), t, Decls)
  }
```

На рис. 9 приведен пример трансляции элемента Resp2, имеющего действие “x = 10; z = Mid;”. Места x и z являются местами переменных, порожденными функцией TranslateExpr().

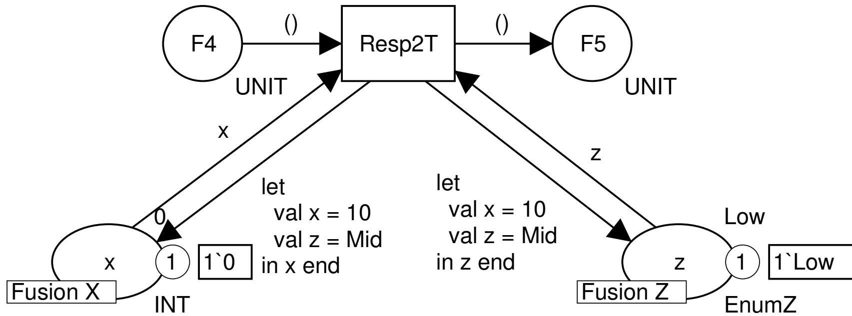

Рис. 9. Трансляция Responsibility

6. ТРАНСЛЯЦИЯ КОНСТРУКЦИЙ АЛЬТЕРНАТИВНОГО ВЫБОРА И ПАРАЛЛЕЛИЗМА

В данном разделе рассмотрим трансляцию элементов, моделирующих параллелизм (And-Fork, And-Join) и альтернативный выбор (Or-Fork, Or-Join). Отметим, что, несмотря на то, что конструкции являются парными, они не обязаны быть правильно вложенными или упорядоченными в каком-либо смысле.

6.1. And-Fork

And-Fork – элемент UCM, представляющий начало параллельного исполнения нескольких (минимум двух) ветвей в сценарии. Его трансляция достаточно проста и интуитивна.

```
TranslateElem(v | v ∈ V & type(v) = AndFork, Decls) → H
{
  fakeIn = u | u ∈ in(v), type(u) = FakePathNode
  in = place( uid(fakeIn) )
```

```
   fakeOuts = { u | u ∈ out(v), type(u) = FakePathNode }
   outs = { place( uid(fakeOut) ) | fakeOut ∈ fakeOuts }

   t = trans( uid(v)+"T" )
   H := < {t}, {in} ∪ outs, { arc(in, t) } ∪ { arc(t, out) | out ∈ outs }, Decls >
}
```

На рис. 10 приведен пример трансляции элемента AndFork, имеющего две исходящие ветви с элементами F4 и F7.

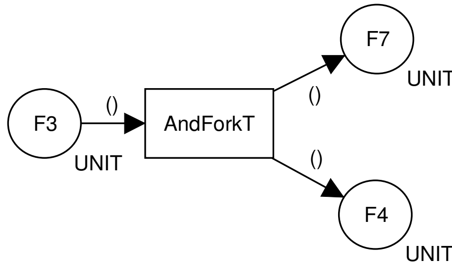

Рис. 10. Трансляция AndFork

6.2. And-Join

And-Join моделирует точку синхронизации альтернативных или параллельных путей исполнения в сценарии. Чтобы исполнение продолжилось далее элемента And-Join, другими словами, по дуге, исходящей из этого элемента, необходимо, чтобы исполнение достигло этого элемента по всем входным дугам.

```
TranslateElem(v | v ∈ V & type(v) = AndJoin, Decls) → H
{
   fakeIns = { u | u ∈ in(v), type(u) = FakePathNode }
   ins = { place( uid(fakeIn) ) | fakeIn ∈ fakeIns }
   fakeOut = u | u ∈ out(v), type(u) = FakePathNode
   out = place( uid(fakeOut) )

   t = trans( uid(v)+"T" )
   H := < {t}, ins ∪ {out}, { arc(in, t) | in ∈ ins } ∪ { arc(t, out) }, Decls >
}
```

На рис. 11 приведен пример трансляции элемента AndJoin с дочерней диаграммы Plugin. Этот элемент является точкой синхронизации двух путей, на которые были добавлены элементы PF2 и PF5 типа FakePathNode.

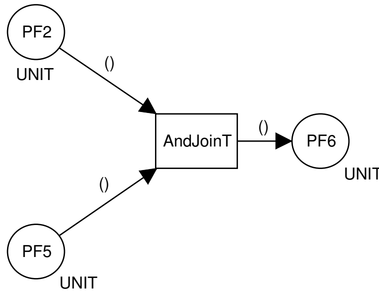

Рис. 11. Трансляция AndJoin

6.3. Or-Fork

Or-Fork моделирует условный выбор между минимум двумя альтернативными путями исполнения сценария. Каждой альтернативной исходящей дуге сопоставлено условие, выраженное пометкой на этой дуге. Когда исполнение достигает элемента Or-Fork, вычисляются условия на дугах. Если ровно одно условие истинно, то исполнение продолжается по соответствующему пути. Будем называть совокупность таких условий корректными. В других случаях исполнение не продолжается за элемент Or-Fork и генерируется предупреждение. В случае если условие не указано хотя бы для одной исходящей дуги, исходная UCM спецификация считается некорректной и трансляции не подвергается.

Несмотря на то, что во многих случаях корректность условий можно проверить статически, в том числе и до трансляции, для общности будем проверять ее динамически. То есть будет введено специальное служебное место warn цвета UNIT, наличие фишек на котором будет сигнализировать о том, что при одном из предыдущих прохождений элемента Or-Fork была нарушена корректность условий. Для проверки корректности условий вводится определение новой функции, которая возвращает true в случае, если для данного элемента Or-Fork и значений переменных совокупность условий является корректной. Определение этой функции в алгоритме ниже хранится в переменной checkerDecl.

При трансляции элементов типа Or-Fork и Timer семантика атрибутов исходящих дуг probability, обозначающих вероятность выбора пути, не рассматривается.

```
TranslateElem(v | v ∈ V & type(v) = OrFork, Decls) → H
{
  warn = place( uid(v)+"_OrForkWarnings" )

  fakeIn = u | u ∈ in(v), type(u) = FakePathNode
  in = place( uid(fakeIn) )
  fakeOuts = { u | u ∈ out(v), type(u) = FakePathNode }
  outs = { place( uid(fakeOut) ) | fakeOut ∈ fakeOuts }

  // транслированные условия исходящих дуг
  conds = { ExprToML( expr((v, u)) ) | u ∈ fakeOuts }
  // список условий на CPN ML через запятую
  condsList = join(",", conds)
  // множество используемых в условиях имен переменных
  varNames = { name | < name, _, _ > ∈ rvalues( expr((v, u)) ),
                          u ∈ fakeOuts}
  // список используемых имен переменных через запятую
  varsList = join(",", varNames)
  id = uid(v)

  // определение функции, определяющей корректность условий
  // на исходящих дугах элемента с идентификатором id.
  // условия корректны, если одно и только одно из них истинно.
  checkerDecl = "fun CheckOrFork_$id($varsList) =
                          let
                              val conditions = [$condsList]
                          in
                              length (rmall false conditions) = 1
                          end;"

  t = trans( uid(v)+"T" )
  H :=   <  {t}, {warn, in} ∪ outs,
            { arc(in, t) } ∪        // входящая дуга
            { arc(t, out, c) |      // исходящие дуги
                out = place( uid(u) ) ∈ outs,
                // с условиями и проверкой корректности
                c  =  "    if $cond andalso CheckOrFork_$id($varsList)
```

```
                        then 1`()
                        else empty",
                cond = ExprToML( expr((v, u)) ) ∈ conds } ∪
            { arc(t, warn, c) |     // исходящая дуга к месту warn,
                // в которое попадает фишка, только если условия
                // не являются корректными
                c  =   "    if CheckOrFork_$id($varsList)
                        then empty
                        else 1`()" },
            // к определениям также добавляется новая функция
            Decls ∪ {checkerDecl} >
    // к фрагменту CPN добавляются необходимые места переменных
    H := H ∪ { TranslateExpr(expr((v, u)), t, Decls) | u ∈ fakeOuts }
}
```

На рис. 12 приведен пример трансляции элемента OrFork с условиями на исходящих дугах [!y] и [y]. Определение сгенерированной функции CheckOrFork_OrFork(y) приведено в Приложении 2.

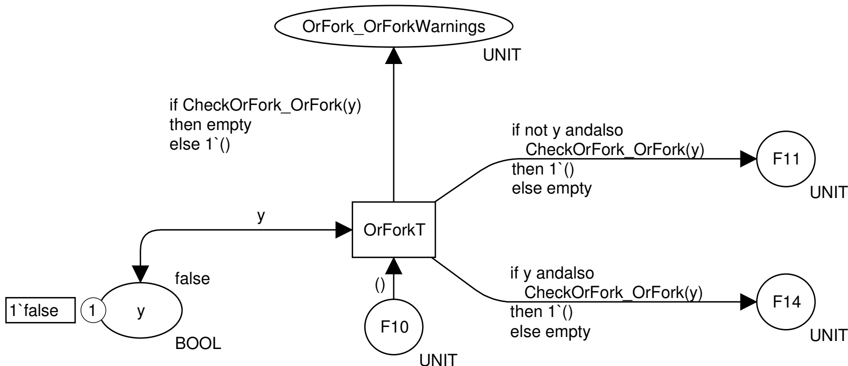

Рис. 12. Трансляция OrFork

6.4. Or-Join

Or-Join моделирует точку слияния альтернативных или параллельных путей исполнения сценария. Or-Join не подразумевает какой-либо синхронизации. При достижении элемента Or-Join по одной из входящих дуг исполнение продолжается за этот элемент по исходящей дуге.

```
TranslateElem(v | v ∈ V & type(v) = OrJoin, Decls) → H
{
  fakeIns = { u | u ∈ in(v), type(u) = FakePathNode }
  ins = { place( uid(fakeIn) ) | fakeIn ∈ fakeIns }
  fakeOut = u | u ∈ out(v), type(u) = FakePathNode
  out = place( uid(fakeOut) )

  ts = { trans( uid(v)+"T"+uid(fakeIn) ) | fakeIn ∈ fakeIns }
  arcs = { arc( place(uid(fakeIn)), trans(uid(v)+"T"+uid(fakeIn)) ),
           arc( trans(uid(v)+"T"+uid(fakeIn)), out ) | fakeIn ∈ fakeIns }
  H := < ts, ins ∪ {out}, arcs, Decls >
}
```

На рис. 13 приведен результат трансляции элемента OrJoin с двумя входящими дугами. Заметим, что при трансляции элементов этого типа получается более одного перехода по числу входящих дуг.

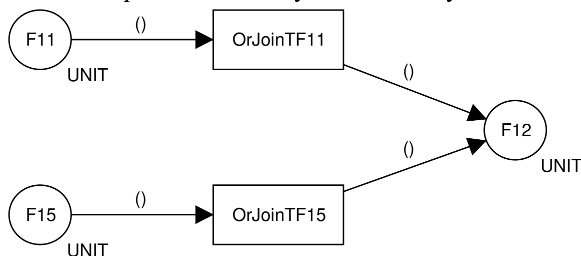

Рис. 13. Трансляция OrJoin

7. ТРАНСЛЯЦИЯ КОНСТРУКЦИЙ ОЖИДАНИЯ

Рассмотрим трансляцию двух конструкций ожидания – Waiting Place и Timer. Как было указано ранее, в UCM нет понятия времени как такового, и при трансляции не используются временные сети Петри.

Оба типа элементов имеют атрибут waitKind, определяющий характер обработки приходящих сигналов. Допустимые значения этого атрибута – Transient и Persistent. Если waitKind = Transient, то соответствующий элемент способен регистрировать только те события, которые приходят, когда другой путь уже ожидает продолжения исполнения в элементе

Waiting Place или Timer. Если waitKind = Persistent, то регистрируются все приходящие события.

В данной работе рассматриваются только элементы с атрибутом waitKind равным Persistent. Отметим лишь, что случаи с waitKind = Transient могут быть смоделированы посредством введения дополнительного перехода, обнуляющего активацию и расположенного непосредственно перед «основным» переходом с именем uid(v) + "T" в алгоритме ниже. К сожалению, такой подход имеет определенные недостатки, например, неатомарное срабатывание элемента в полученной CPN модели.

7.1. Waiting Place

Waiting Place представляет точку в сценарии, где продолжение сценария зависит от выполнения некоторого условия или прибытия сигнала о некотором событии через Connect. Исполнение продолжается за Waiting Place, если условие, сопоставленное этому элементу, истинно, либо если было зарегистрировано хотя бы одно событие, то есть активация оказалась положительна. Стандарт специфицирует, что в обратном случае, т. е. если условие ложно, и сигнал никогда не приходит, генерируется предупреждение. Считаем, что при трансляции этот случай является штатной ситуацией, наличие которой можно будет определить при анализе полученной модели.

```
TranslateElem(v | v ∈ V & type(v) = WaitingPlace, Decls) → H
{
   fakeIn = u | u ∈ in(v), type(u) = FakePathNode
   fakeOut = u | u ∈ out(v), type(u) = FakePathNode
   in = place( uid(fakeIn) )
   out = place( uid(fakeOut) )

   if hasConnect(v) {
       t = trans( uid(v)+"T", "[" + ExprToML(expr(v)) + " orelse act > 0]" )
   }  else {
       t = trans( uid(v)+"T", "[" + ExprToML(expr(v)) + "]" )
   }

   H := < {t}, {in, out}, { arc(in, t), arc(t, out) }, Decls >
   H := H     ∪ AddConnectTrigger(v, t, Decls, IN)
              ∪ TranslateExpr(expr(v), t, Decls)
}
```

На рис. 14 приведен пример трансляции элемента Wait с дочерней диаграммы Plugin с условием продолжения [v = 1], которое было преобразовано в охранное условие.

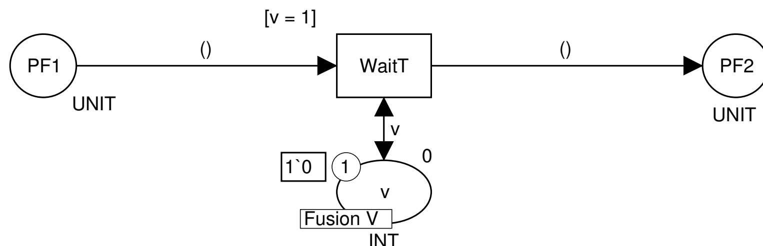

Рис. 14. Трансляция WaitingPlace, не смежного с Connect

7.2. Timer

Timer является наследником Waiting Place в метамодели UCM. Также он имеет черты Or-Fork, так как имеет два альтернативных исходящих пути, называемых release и timeout. Timer является специализацией Waiting Place, для которой продолжение сценария зависит от выполнения условия продолжения на release пути, прибытия сигнала через Connect или наступления тайм-аута, т. е. выполнения условия на timeout пути. Так же, как и для Or-Fork, семантика атрибутов probability не рассматривается.

Для элементов Timer наличие исходящего timeout пути является опциональным. В случае если такой путь отсутствует, элемент Timer становится функционально эквивалентным элементу Waiting Place. Поэтому элементы Timer без timeout пути будем транслировать как элементы Waiting Place.

Исполнение может продолжиться далее элемента Timer по release пути, если условие release пути истинно, либо условия release и timeout путей ложны, и активация положительна. Исполнение может продолжиться по timeout пути, если условие release пути ложно, а условие timeout пути истинно, либо условия release и timeout путей ложны и активация отсутствует, т. е. равна нулю.

Для описания алгоритма требуется способ различать исходящие пути. Введем для этого следующие две функции:

isReleasePath(e) – истина, если указанная дуга e является release путем элемента Timer;

isTimeoutPath(e) – истина, если указанная дуга e является timeout путем элемента Timer.

Ниже приведено описание алгоритма трансляции элемента Timer для случая наличия timeout пути. Иначе следует использовать реализацию функции для элементов типа Waiting Place.

```
TranslateElem(v | v ∈ V & type(v) = Timer, Decls) → H
{
   fakeIn = u | u ∈ in(v), type(u) = FakePathNode
   fakeReleasePath = u | u ∈ out(v), type(u) = FakePathNode,
                          isReleasePath( (v, u) )
   fakeTimeoutPath = u | u ∈ out(v), type(u) = FakePathNode,
                          isTimeoutPath( (v, u) )
   in = place( uid(fakeIn) )
   rpath = place( uid(fakeReleasePath) )
   topath = place( uid(fakeTimeoutPath) )
   releaseCond = ExprToML( expr((v, fakeReleasePath)) )
   timeoutCond = ExprToML( expr((v, fakeTimeoutPath)) )

   // конструирование условий release и timeout путей в соответствии
   // с семантикой элемента в зависимости от смежности с Connect
   if hasConnect(v) {
       timeoutPathCond = "  if $releaseCond = false andalso
                                ( $timeoutCond = true orelse act = 0 )
                            then 1`()
                            else empty"
       releasePathCond = "  if $releaseCond = true orelse
                                ( $releaseCond = false andalso
                                $timeoutCond = false andalso
                                act > 0 )
                            then 1`()
                            else empty"
   } else {
       timeoutPathCond = "  if $releaseCond = false andalso
                                $timeoutCond = true
                            then 1`()
                            else empty"
       releasePathCond = "  if $releaseCond = true
                            then 1`()
                            else empty"
   }
```

```
   t = trans( uid(v)+"T" )
   H :=   <  {t}, {in, rpath, topath},
             { arc(in, t),    arc(t, rpath, releasePathCond),
                              arc(t, topath, timeoutPathCond) },
             Decls >
   H := H    ∪ AddConnectTrigger(v, t, Decls, IN)
             ∪ TranslateExpr(expr(v), t, Decls)
}
```

На рис. 15 приведен пример трансляции элемента Timer, смежного с элементом Connect2 и имеющим условие release пути [z = High], а условие timeout пути [z = Low].

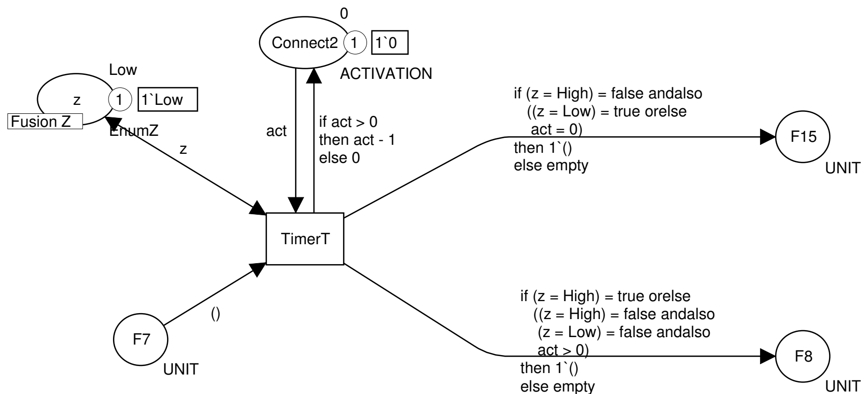

Рис. 15. Трансляция Persistent Timer

8. ТРАНСЛЯЦИЯ КОНСТРУКЦИЙ ДЕКОМПОЗИЦИИ

Рассмотрим трансляцию одной из конструкций декомпозиции, называемой Static Stub, а также то, как изменяется алгоритм трансляции для элементов Plug-in maps. Plug-in maps могут быть обозначены как существующие в единственном экземпляре через атрибут singleton. Рассмотрим только трансляцию Static Stub с дочерними диаграммами, не обладающими атрибутом singleton, т.к. это наиболее простая и вместе с тем часто используемая конструкция декомпозиции, не требующая усложнения устройства транспорта сигналов, такого, как, например, использование стека иденти-

фикаторов потока управления вместо простого типа UNIT. Также считаем, что для всех элементов типа Static Stub в исходной UCM-спецификации задана одна дочерняя диаграмма.

В такой постановке возможна подстановка Plug-in maps в соответствующие элементы Stub. Описание такого процесса дано в секции 8.3.1 «Flattened UCM models» стандарта [2]. Исходя из этого, можно было бы ограничиться трансляцией уже «плоской» модели UCM, то есть модели без элементов типа Stub. Однако для иллюстрации общего подхода к трансляции конструкций декомпозиции и сохранения первоначальной структуры модели будем транслировать иерархические UCM модели в иерархические раскрашенные сети Петри.

8.1. Static Stub

Для описания трансляции иерархических моделей UCM введем следующие дополнительные функции.

plugin(v) → G позволяет получить дочернюю диаграмму для заданного элемента v ∈ V | isStub(v).

inBinding(e) → w возвращает элемент w ∈ V(plugin(v)) | type(w) = StartPoint с дочерней диаграммы, соответствующий данной входящей дуге e = (u, v) | isStub(v) & u ∈ in(v).

outBinding(e) → w возвращает элемент w ∈ V(plugin(v)) | type(w) = EndPoint с дочерней диаграммы, соответствующий данной исходящей дуге e = (v, u) | isStub(v) & u ∈ out(v).

Также введем следующие дополнительные конструкторы для раскрашенных сетей Петри.

t = **subst**(name, submodule) – новый подстановочный переход (substitution transition) с уникальным именем name и связанным с ним модулем submodule.

p = **socket**(name, port, colset=UNIT, init=empty) – новый сокет (входное или выходное место подстановочного перехода) с уникальным именем name, цветом colset и начальной разметкой init, связанный с портом port.

p = **port**(name, colset=UNIT) – новый порт (входное или выходное место модуля) с уникальным именем name, цветом colset и начальной разметкой init.

a = **arc**(t, p) или a = **arc**(p, t) – новая дуга от подстановочного перехода к сокету или, наоборот, от сокета к подстановочному переходу. Заметим, что на таких дугах выражения недопустимы.

Заметим, что если p = port(pname, pcolset) и s = socket(sname, p, scolset, sinit), то pcolset = scolset. Аналогично должны выполняться и другие ограничения синтаксической корректности CPN модели.

Ниже приведен алгоритм трансляции элементов Static Stub. Детализация алгоритма трансляции элементов на дочерних диаграммах приведена в следующем подразделе. Для алгоритма объединения транслированных фрагментов в единую CPN модель сокеты эквивалентны обычным местам с тем же именем. В объединенной модели такие места являются сокетами.

В этом разделе считаем, что фрагмент CPN модели является непрозрачным типом данных, конструировать значения которого можно прежним образом, как кортеж <T, P, A, D>, но только в рамках одного модуля или верхнего уровня CPN. В алгоритме ниже переменные H и P имеют такой тип. Явное задание иерархии модулей для значений данного типа при этом абстрагируется посредством использования конструктора подстановочных переходов subst().

Так как одна дочерняя диаграмма может использоваться в нескольких элементах типа Stub, то для соответствующего модуля CPN необходимо создать порты для всех элементов Start Point и End Point, которые связаны с входными или выходными дугами какого-либо элемента Stub. В силу этого могут образоваться «висячие» Start Points и End Points, и для них нужно будет создать отдельные сокеты. Более того, имеет смысл конструкция, когда несколько входных дуг соответствуют одному элементу Start Point на дочерней диаграмме. Такая конструкция аналогична элементу Or-Join перед элементом Stub. Для выходных дуг аналогичная конструкция не имеет смысла и не рассматривается.

```
TranslateElem(v | v ∈ V & isStub(v) & type(v) = StaticStub, Decls) → H
{
   G = plugin(v)
   // элементы FakePathNode на входящих и исходящих дугах
   fakeIns = { u | u ∈ in(v), type(u) = FakePathNode }
   fakeOuts = { u | u ∈ out(v), type(u) = FakePathNode }
   // элементы Start и End Point, привязанные на дочерней диаграмме
   boundStarts = { w | w ∈ V(G), w = inBinding((u, v)), u ∈ fakeIns }
   boundEnds = { w | w ∈ V(G), w = outBinding((v, u)), u ∈ fakeOuts }
   // все элементы Start и End Point с дочерней диаграммы
   allStarts = { w | w ∈ V(G), w = inBinding((u, v)) ∀u }
   allEnds = { w | w ∈ V(G), w = outBinding((v, u)) ∀u }
   // «висячие» элементы Start и End Point с дочерней диаграммы
   hangingStarts = allStarts / boundStarts
```

```
   hangingEnds = allEnds / boundEnds
   // элементы Start Point на дочерней диаграмме, связанные более
   // чем с одной входящей дугой v
   multiStarts = { w | w ∈ boundStarts,
                       w = inBinding((u1, v)), w = inBinding((u2, v)), u1 != u2 }
   // элементы Start Point на дочерней диаграмме, связанные ровно с
   // одной входящей дугой v
   normalStarts = boundStarts / multiStarts

   // транслируем плагин и создаем его подстановочный переход
   P = TranslatePlugin(G, Decls)
   t = subst( uid(v) + "T", P )

   // сокеты и места для различных классов элементов Start Point
   hangingIns = { socket( uid(v) + "S_" + uid(w), w, UNIT, () ) |
                       w ∈ hangingStarts }
   normalIns = { socket( uid(u), w ) |
                       w ∈ normalStarts, w = inBinding((u, v)) }
   collectiveIns = { socket( uid(v) + "S_" + uid(w), w ) | w ∈ multiStarts }
   multiIns = { place( uid(u) ) | w ∈ multiStarts, w = inBinding((u, v)) }
   socketIns = hanginIns ∪ normalIns ∪ collectiveIns
   ins = socketIns ∪ multiIns

   // и переходы с дугами для них
   multiInTr = { trans( uid(u) + "T" ) |
                       w ∈ multiStarts, w = inBinding((u, v)) }
   inArcs =  { arc(in, t) | in ∈ socketIns } ∪
             { arc(mIn, inTr), arc(inTr, cIn) |
                 inTr = trans( uid(u) + "T" ) ∈ multiInTr,
                 cIn = socket( uid(v) + "S_" + uid(w), w ) ∈ collectiveIns,
                 mIn = place( uid(u) ),
                 w ∈ multiStarts, w = inBinding((u, v)) }

   // сокеты для связанных и «висячих» элементов End Point
   normalOuts = { socket( uid(u), w ) |
                       w ∈ boundEnds, w = outBinding((v, u)) }
   hanginOuts = { socket( uid(v) + "S_" + uid(w), w ) | w ∈ hangingEnds }
   outs = normalOuts ∪ hangingOuts
   outArcs = { arc(t, out) |  out ∈ outs }    // а также переходы для них
   H := P ∪ < {t} ∪ multiInTr, ins ∪ outs, inArcs ∪ outArcs, Decls >
}
```

8.2. Трансляция дочерних диаграмм

При трансляции дочерних диаграмм трансляция элементов Start Point и End Point может отличаться. Кроме того, места переменных, встречающиеся более чем в одном модуле CPN, но имеющие одинаковые имена, при объединении транслированных фрагментов становятся совмещенными, также называемые fusion places. Считаем, что функции DefineColors(G), DefineConstants(G), DefineVars(G) и AddFakeNodes(G) при трансляции иерархической UCM модели способны обработать все вершины и ребра иерархического графа, а не только его верхнего уровня. В таком случае определение функции Translate(G) сохраняется, с поправкой на инициализацию графа H := <∅, ∅, ∅, Decls>, означающую создание пустого иерархического графа с набором определений Decls. Определим функцию трансляции дочерней диаграммы.

```
TranslatePlugin(G, Decls) → P
{
   // элементы Start и End Point, связанные с каким-либо Stub
   bound = { w | w ∈ V(G),
                   w = inBinding((u, v)) ∨ w = outBinding((v, u))
                   ∀u ∀v | isStub(v) }

   P := < ∅, ∅, ∅, Decls >
   // все элементы транслируются так же, как и ранее
   ∀ v ∈ V(G) / bound | isPrimaryElem(v)
   {
       H := H ∪ TranslateElem(v, Decls)
   }
   // кроме связанных, для которых используется альтернативный
   // алгоритм трансляции TranslatePluginElem()
   ∀ v ∈ bound
   {
       H := H ∪ TranslatePluginElem(v, Decls)
   }
}
```

В этом алгоритме переменная bound содержит все элементы Start Point и End Point данной дочерней диаграммы, которые связаны с дугами какихлибо элементов типа Stub. Для таких элементов используется альтернативная версия алгоритма трансляции, приведенная ниже.

```
TranslatePluginElem(v | v ∈ V & type(v) = StartPoint, Decls) → H
{
   in = port( uid(v) )
   fakeOut = u | u ∈ out(v), type(u) = FakePathNode
   out = place( uid(fakeOut) )

   if hasConnect(v) {
        t = trans( uid(v) + ”T”, “[“ + ExprToML(expr(v)) + “, act > 0]” )
   } else {
        t = trans( uid(v) + “T”, “[“ + ExprToML(expr(v)) + “]” )
   }

   H := < {t}, {in, out}, { arc(in, t), arc(t, out) }, Decls >
   H := H     ∪ AddConnectTrigger(v, t, Decls, IN, “act-1”)
              ∪ TranslateExpr(expr(v), t, Decls)
}

TranslatePluginElem(v | v ∈ V & type(v) = EndPoint, Decls) → H
{
   endPostCond = place( uid(v) + “_PC”, SATISFIED, empty )
   fakeIn = u | u ∈ in(v), type(u) = FakePathNode
   in = place( uid(fakeIn) )
   out = port( uid(v) )

   if expr(v) = Ø {
        postCond = true
   } else {
        postCond = ExprToML(expr(v))
   }

   t = trans( uid(v)+”T” )

   H :=   <  {t}, {in, endPostCond, out},
              { arc(in, t), arc(t, endPostCond, postCond), arc(t, out) },
              Decls  >
   H := H     ∪ AddConnectTrigger(v, t, Decls, OUT)
              ∪ TranslateExpr(expr(v), t, Decls)
}
```

Несложно заметить, что отличие этих версий алгоритма трансляции в том, что для элементов Start Point и End Point всегда создается пустое место

цвета UNIT, входное или выходное соответственно. Эти места используются как порты данного модуля.

8.3. Пример

Рассмотрим примеры простого использования элемента Static Stub и особенностей трансляции дочерней диаграммы. На рис. 16 – 18 представлены результаты трансляции элементов Stub, PStart2 и PEnd.

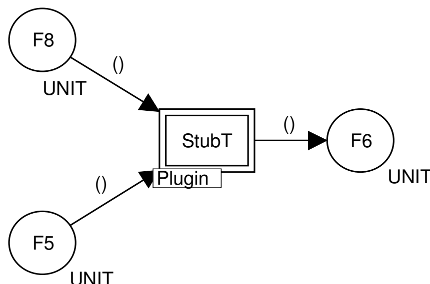

Рис. 16. Трансляция элемента Stub без висячих Start Point

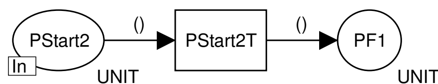

Рис. 17. Трансляция элемента PStart2 - Start Point в дочерней диаграмме

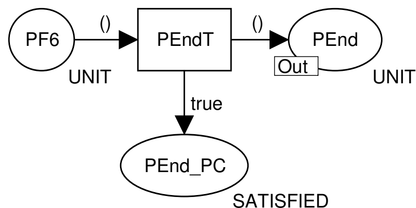

Рис. 18. Трансляция элемента PEnd - End Point в дочерней диаграмме

9. ЗАКЛЮЧЕНИЕ

Графическая нотация Use Case Maps является выразительным средством описания функциональных требований к программным системам и протоколам. Она позволяет описывать сценарии как совокупность причинно-следственных связей между событиями в системе, опционально отображенными на структуру ее компонентов. В настоящей работе описаны алгоритмы трансляции UCM-спецификаций в раскрашенные сети Петри без времени. Разработаны алгоритмы для трансляции большинства конструкций UCM, описанных в стандарте, за исключением сложных конструкций декомпозиции, некоторых редко используемых вариантов семантики конструкций ожидания и семантики компонентов. Алгоритмы были опробованы на нескольких тестовых примерах.

Для анализа полученных CPN моделей предлагается использовать встроенные средства пакета CPN Tools [10] и новый верификатор раскрашенных сетей Петри.

Планируется расширить транслируемое подмножество UCM за счет новых типов конструкций декомпозиции и расширений для обработки исключительных ситуаций [5, 7]. Также планируется реализация прототипа транслятора из формата редактора jUCMNav [9] в формат пакета CPN Tools.

СПИСОК ЛИТЕРАТУРЫ

1. Specification and Description Language (SDL). – CCITT, Recommendation Z.100, 1988.
2. User Requirements Notation (URN) – Language definition. – ITU-T, Recommendation Z.151, 2008.
3. Jensen K., Kristensen L.M. Coloured Petri Nets: Modelling and Validation of Concurrent Systems. // Springer 2009.
4. Jensen K., Kristensen L.M., Wells L. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. // Int. J. on Software Tools for Technology Transfer 9.- 2007.- P. 213-254.
5. Baranov S., Kotlyarov V., Weigert T. Verifiable Coverage Criteria for Automated Testing. // SDL 2011, LNCS 7083.- 2011.- P. 79-89.
6. Baranov S.N., Drobintsev P.D., Kotlyarov V.P., Letichevsky A.A. The Technology of Automated Verification and Testing in Industrial Projects. // Proc. IEEE Russia Northwest Section, 110 Anniversary of Radio Invention Conference.- IEEE Press, St.Petersburg, 2005.- P. 81-89.

7. Amyot D., Weiss M., Logrippo L. Generation of Test Purposes from Use Case Maps. // Computer Networks 49(5).- 2009.- P. 643-660.
8. Котляров В.П., Дробинцев П.Д. Технология разработки качественного программного продукта на основе интеграции верификации и тестирования. // Международный семинар «Семантика, спецификация и верификация программ: теория и приложения» под ред. Непомнящего В.А. и Соколова В.А. – Казань, 2010. – С. 62-69.
9. Официальный сайт интегрированной среды разработки jUCMNav – 2012 [Электронный ресурс]. – URL: http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome .
10. Официальный сайт средства разработки и анализа раскрашенных сетей Петри CPN Tools – 2012 [Электронный ресурс]. – URL: http://cpntools.org/ .

ПРИЛОЖЕНИЕ 1. СПЕЦИФИКАЦИЯ ПРИМЕРА НА ЯЗЫКЕ UCM

Ниже приведен синтетический пример UCM-спецификации, используемый в данной работе. Результат его трансляции в CPN приведен в Приложении 2.

Спецификация состоит из двух диаграмм – диаграммы верхнего уровня (Global) и дочерней диаграммы (Plugin). В спецификации, кроме обозначенных элементов, также присутствует один элемент типа Empty Point и два элемента типа Connect. Элемент Stub связан с дочерней не-singleton диаграммой Plugin следующим образом: IN1 ⇔ PStart1, IN2 ⇔ PStart2, OUT1 ⇔ PEnd.

Переменные имеют следующие типы и начальные значения:

x : Integer = 0;
y : Boolean = false;
z : Enumeration EnumZ { Low, Mid, High } = Low;
v : Integer = 0.

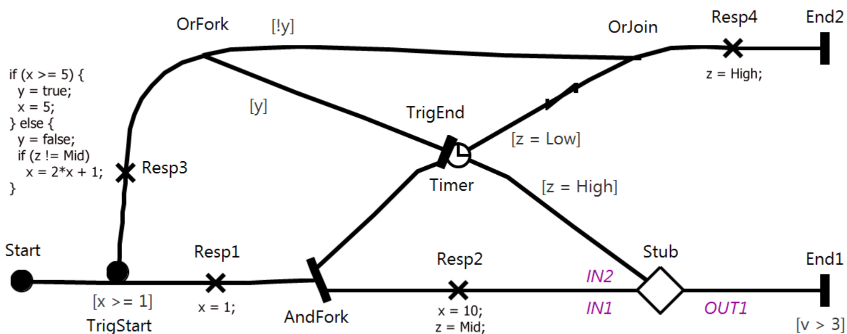

Рис. 19. UCM диаграмма верхнего уровня Global

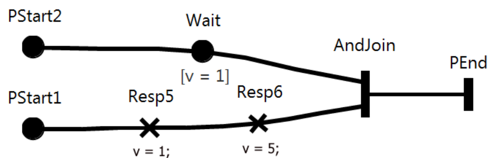

Рис. 20. Дочерняя UCM диаграмма Plugin

ПРИЛОЖЕНИЕ 2. ТРАНСЛЯЦИЯ ПРИМЕРА В РАСКРАШЕННУЮ СЕТЬ ПЕТРИ

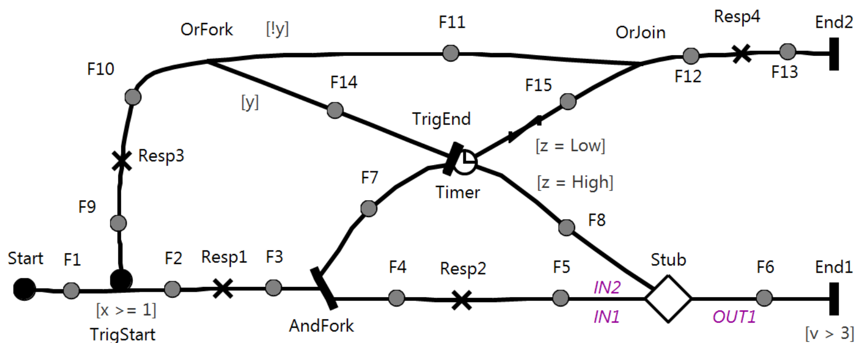

Рис. 21. Добавление элементов FakePathNode на диаграмму Global

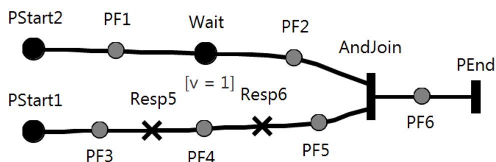

Рис. 22. Добавление элементов FakePathNode на диаграмму Plugin

На следующих страницах приведен результат трансляции UCM-спецификации в CPN. На приведенных иллюстрациях для упрощения восприятия места переменных `x`, `z` и `v` являются совмещенными (fusion places). В результате работы алгоритма только место переменной `v` является совмещенным, т.к. она используется на разных диаграммах. Также выражения на дугах, смежных с местом `Resp3T`, были заменены функциями `Resp3X()` и `Resp3Y()` по той же причине. Эти функции задают правила изменения значений переменных `x` и `y` при прохождении элемента `Resp3`. Они описаны в разделе 4.4 среди примеров трансляции действий.

Рис. 23. Транслированная диаграмма Global, левая часть

Рис. 24. Транслированная диаграмма Plugin, левая часть

OrJoinTF11 () F12 () Resp4T () F13 () End2T
UNIT UNIT
z
let val z = High
in z end
true
TrigEndT true TrigEnd_PC
Low
z 1 1`Low
Fusion Z EnumZ
End2_PC
SATISFIED
SATISFIED
act act + 1
()
OrJoinTF15
0
Connect2 1 1`0
ACTIVATION
()
if (z = High) = false andalso
((z = Low) = true orelse
act = 0)
then 1`()
else empty
act
if act > 0
then act - 1
else 0
F15
UNIT
TimerT
End1_PC
SATISFIED
()
if (z = High) = true orelse
((z = High) = false andalso
(z = Low) = false andalso
act > 0)
then 1`()
else empty
F7
UNIT
F8
UNIT
if v > 3
then true
else false
()
()
AndForkT () F4 () Resp2T () F5 () StubT () F6 () End1T
UNIT UNIT Plugin UNIT
x
z
v
let
val x = 10
val z = Mid
in x end
let
val x = 10
val z = Mid
in z end
0
Low
0
x 1 1`0
Fusion X INT
z 1 1`Low
Fusion Z EnumZ
v 1 1`0
Fusion V INT

Рис. 25. Транслированная диаграмма Global, правая часть

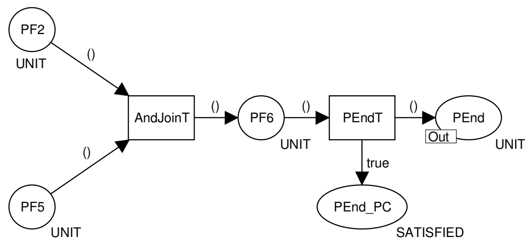

Рис. 26. Транслированная диаграмма Plugin, правая часть

Ниже приведены все определения, используемые в данной CPN модели.

```
Standard Declarations:
  colset UNIT = unit;
  colset INT = int;
  colset BOOL = bool;
UCM Types:
  colset SATISFIED = bool;
  colset ACTIVATION = subset INT by (fn x => x >= 0);
Model Types:
  colset EnumZ = with Low | Mid | High;
Constants:
  val x_pre = 0;
  val y_pre = false;
  val z_pre = Low;
  val v_pre = 0;
Variables:
  var act : ACTIVATION;
  var x : INT;
  var y : BOOL;
  var z : EnumZ;
  var v : INT;
Functions:
  fun CheckOrFork_OrFork(y) =
      let
          val conditions = [y, not y]
      in
          length (rmall false conditions) = 1
      end;
  fun Resp3X   // описана в разделе 4.4
  fun Resp3Y   // описана в разделе 4.4
```

Н.В. Визовитин, В.А. Непомнящий

АЛГОРИТМЫ ТРАНСЛЯЦИИ UCM-СПЕЦИФИКАЦИЙ В РАСКРАШЕННЫЕ СЕТИ ПЕТРИ

Препринт
168

Рукопись поступила в редакцию 05.09.2012
Редактор Т.М. Бульонкова
Рецензент Т.Г. Чурина

Подписано в печать 08.10.2012
Формат бумаги 60 × 84 1/16 Объем 3,4 уч.-изд.л., 3,75 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42