УДК 519.7 + 004.436.4

А.Б. Годлевский

Инсерционная семантика параллельных процедурных конструктов языка *UCM*

Предложено математическое описание семантики процедурного конструкта *Stub* в языке параллельного моделирования *UCM* с применением инсерционного подхода, основанного на взаимодействии сред и погруженных в них агентов.

The mathematical description of the semantics of a procedural construct Stub for the parallel modeling language UCM is suggested. This description used the insertion approach that is based on the interaction of environments and agents inserted in them.

Запропоновано математичний опис семантики процедурного конструкту *Stub* у мові паралельного моделювання *UCM* із застосуванням інсерційного підходу, основаного на взаємодії середовищ та занурених у них агентів.

1. Введение. Параллельные вычисления и языки параллельного программирования, по-видимому, – одно из тех направлений в развитии программирования, которое может служить ответом на бурное развитие электронной инженерии. Особенно актуально развитие высокоуровневых языков и языков, приближенных к инженерным разработкам. В частности, в последние десятилетия получили развитие языки с графическим представлением, позволяющие в удобных для инженеров формах проводить разработку многокомпонентных программных систем. Их представителями могут служить языки *MSC* [1], *SDL* [2], *UML* [3], *UCM* [4] и др.

В настоящей статье[1] рассмотрена семантика исполнения процедурного конструкта *стаб* (*stub*) в языке *UCM*. Описание семантики выполнено в рамках инсерционного подхода [5] к вычислениям в многокомпонентных средах. Суть инсерционной модели состоит в том, что поведение отдельных компонент (агентов) описывается на уровне алгебры поведений [6, 7], а их взаимодействие использует функцию погружения. Посредством этой функции задается взаимодействие агентов и сред, соответствующих разным уровням представления проектируемых систем. Средства, используемые в инсерционном моделировании, позволяют выразить все те особенности модели вычислений, что была предложена для языка *UCM* [4] и в работах [8–12] основных разработчиков этого языка. Отличием настоящего описания семантики *UCM* есть использование математического подхода к описанию процессов вызова процедур, специфицируемых ими вычислений и процессов завершения процедур.

Автор статьи в значительной степени придерживается терминологии, принятой в языке *UCM*, включая использование оригинальных терминов. Отчасти это сделано для того, чтобы подчеркнуть связь с этим языком, а отчасти потому, что адекватная терминология в русском языке отсутствует. Ключевые понятия, используемые здесь – карта как поведенческий сценарий и стаб как конструкт для вызова процедуры.

Стаб (конструкт класса *Stub*) ссылается на совокупность вложенных карт (*plug-in map* или плагинов), перечисленных в описании стаба (структуры *PluginBinding*). Коммутация стартовых и конечных вершин путей вложенных карт с входными и выходными путями стаба задается специальными структурами *InBinding* и *OutBinding*. Более детально семантика будет представлена далее – как иллюстративно на примерах, так и в виде атрибутной транзиционной системы, описанной в рамках инсерционного подхода.

Отметим, что для понимания языка *UCM* удобна модель сетей Петри [13], процессы в которых представляются как движение фишек вдоль путей в сетях. Существенное отличие от обычных сетей Петри состоит в том, что процессы в языке *UCM* развиваются над некоторой памятью, используемой как при вычислении условий (конструктов класса *Condition*), так и при выполнении действий (конструктов

Ключевые слова: язык *UCM*, инсерционное программирование.

[1] Работа выполнена при поддержке ДФФД по проекту Ф40.1/004.

класса *Responsibility*). Вычисляемые условия используется внутри некоторых конструктов в качестве барьеров, разрешающих или запрещающих дальнейшее продвижением фишек.

Язык *UCM* наряду с текстовым обладает также развитым графическим представлением. На приведенных рисунках, а также в табл. 2, где специфицируются разные случаи поведения стабов, используются некоторые элементы этой графики, например, пути, их стартовые и конечные вершины, стабы и др. Здесь не приведены все графические элементы и пояснения к ним, однако отметим, что помимо стандарта языка *UCM* [4], где все они детально описаны, в этом тематическом выпуске содержится работа [14], где представлены конструкты *UCM* плоского (не иерархического) уровня. Здесь же добавлен лишь конструкт стаб.

Статья состоит из восьми разделов, включая введение и заключение. Во втором разделе кратко описана инсерционная модель, в третьем и четвертом разделах даны содержательная характеристика стабов и присущие им параллелизмы, в пятом описываются виды стабов. В шестом и седьмом разделах приведены семантика стабов в рамках инсерционной модели и уравнения, реализующие их поведение.

2. Инсерционная модель

Математическую семантику стабов опишем в рамках инсерционной модели, предложенной А.А. Летичевским [5]. Модель состоит из описания среды и агентов; описания функции погружения, посредством которой изменяется состояние среды и погруженных в нее агентов. В общем случае среда, описывающая конкретный *UCM*-проект, – многоуровневая и включает в себя (глобальную) среду верхнего уровня *E* и вложенные в нее среды карт *M*, визитов *V* и компонент *C*, рассматриваемых как агенты, а также процессы *P* как агенты самого низкого уровня. Уровень компонент может отсутствовать. Все указанные среды задаются посредством описания их свойств и атрибутов. Атрибуты самой внешней среды *E* будем называть глобальными, все прочие свойства и атрибуты локализуются на уровнях соответствующих вложенных сред. Доступ к атрибуту *N*, который относится к локальной среде *M*, задается в виде $M_1.M_2. \ldots .M.N$, где префикс $M_1.M_2. \ldots$ отражает вложенность локальных сред для карт, объемлющих данную карту *M*. В дальнейшем, если из контекста понятно, к какой среде относится атрибут, префикс перед ним для краткости будем опускать.

Другая классификация атрибутов, отличная от их локализации по уровням, заключается в их разделении на два непересекающихся класса: пользовательские и управляющие. Пользовательские атрибуты соответствуют переменным проекта *UCM*, введенным его разработчиком. Они используются в условиях и в действиях, где вычисляются их значения. Управляющие атрибуты создаются с целью обеспечить синхронизацию процессов, соответствующую определяемой семантике. Это происходит на этапе построения модели проекта *UCM* – трансляции проекта в систему сред и агентов.

Любой процесс есть либо действием, либо композицией более простых процессов, образованных посредством применения операций префиксинг (точка, отделяющая действие от хвоста процесса) и недетерминированный выбор (представляется символом +, который отличается от обычного знака сложения тем, что его операнды относятся к алгебре поведений). Помимо этих операций алгебра поведений обогащена операцией параллельной композиции $() \parallel ()$ с семантикой интерливинга. Начальное состояние среды – это параллельная композиция $E[M_1[u_1], M_2[u_2], \ldots]$ стартовых точек (конструктов класса *StartPoint*), рассматриваемых как начальные состояния агентов, погруженных в среды карт и, возможно, компонент (если они к ним привязаны), а также множество начальных значений атрибутов. Переходы в системе определяются уравнениями вида:

$$S = a.S_1, \quad S = S_1 + S_2, \quad S = S_1 \parallel S_2,$$

в которых S – это текущее поведение агента, S_1 и S_2 – выражения в алгебре поведений. В качестве *S* рассматривается вершина пути и ассоциированный с ней конструкт языка *UCM*. Первое уравнение описывает переход агента от *S* к поведению S_1 после выполнения действий, задаваемых префиксом *a*. Выражение S_1 в пер-

вом уравнении может отсутствовать, и тогда такой переход интерпретируется как успешное завершение агента и удаление его из среды, т.е. из числа погруженных в нее агентов. Второе уравнение задает альтернативный переход агента к одному из поведений S_1 или S_2. В третьем случае S поведение агента заменяется поведениями S_1 и S_2 уже двух агентов, погруженных в среду. Если использовать терминологию сетей Петри, то уравнение $S = a.S_1$ можно интерпретировать как переход фишки из начала пути S, представленного вершиной a, к началу пути S_1; уравнение $S = S_1 + S_2$ – как продвижение фишки через точку ветвления к началу одной из ветвей; и уравнение $S = S_1 \parallel S_2$ – как раздвоение фишки и переход ее копий к началам обоих путей S_1 и S_2.

Функция погружения (*insertion function*) задается совокупностью правил, описывающих различные случаи взаимодействия между агентами и средами. Поскольку определяемые здесь правила используют фиксированный набор конкретных управляющих атрибутов системы, математическое описание этих правил будет следовать после описания системы атрибутов. В [14] описаны правила *apply* (выполнение действий над атрибутами), *check* (проверка выполнимости условий), *insert* (погружение нового агента в среду). При описании семантики стабов будут определены правила *insertvisit* (образование среды визита), *movein* (переход процесса во вложенную карту) и *moveout* (выход процесса из вложенной карты в карту верхнего уровня).

3. Содержательная характеристика стабов

Описание стаба содержит одну или несколько структур *PluginBinding*, каждая из которых связывает этот стаб в точности с одной картой. Такая структура специфицирует коммутации входных и выходных путей стаба со стартовыми и конечными вершинами путей в соответствующей иерархически подчиненной карте. Коммутация заданной карты M и стаба St задается структурами двух видов, *InBinding* и *OutBinding*, которые будем называть также входными и выходными *привязками* стаба к вложенным картам. Каждая *InBinding* (соответственно, *OutBinding*) структура описывает соединение одного из входных (выходных) путей этого стаба с одной из стартовых (конечных) вершин путей карты M. Входные (выходные) пути стаба представлены вершинами, непосредственно предшествующими стабу (следующими за ним).

Итак, *InBinding* структура стаба – это пара (*in*, *sp*), где *in* – входной путь этого стаба, *sp* – стартовая вершина пути карты. Для стаба St и карты M все пары вида (*in*, *sp*) различаются между собой, задавая частичное взаимно однозначное соответствие между входными путями стаба St и стартовыми вершинами вложенной карты M. Аналогично, *OutBinding* структура – это пара (*ep*, *out*), где *ep* – конечная вершина пути карты, *out* – выходной путь стаба.

На рис. 1 в левой его части изображена структура стаба St с двумя входными путями от вершин S_1 и S_2, непосредственно предшествующими стабу, двумя выходными путями к вершинам T_1 и T_2, непосредственно следующими за стабом, карта M с двумя стартовыми и тремя конечными вершинами. Слева и справа от карты изображены структуры стаба *IB* (*InBinding*), *OB* (*OutBinding*), обеспечивающие соединение входных и выходных путей стаба St со стартовыми и конечными вершинами путей вложенной карты. Понятно, что данных, содержащихся в структурах *IB* и *OB*, достаточно для восстановления путей от S_1 и S_2 к стартовым вершинам путей карты М и от конечных вершин ее путей к вершинам T_1 и T_2. Эти пути, обозначенные пунктирными линиями, используют данные структур *IB* и *OB*.

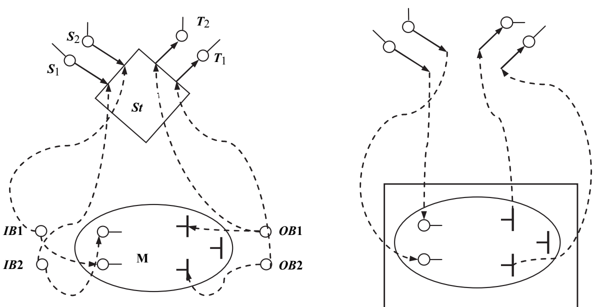

Рис. 1. Схема структуры и динамики стаба с одной вложенной картой

В правой части этого рисунка изображено, каким образом в карту верхнего уровня будет вложена карта M – плагин для стаба St и как в

результате будут развиваться процессы, специфицированные стабом. Прямоугольник, в который заключена карта *M*, рассматривается как промежуточный уровень между исходной картой со стабом *St* и картой *M*. Переход фишки внутрь этого прямоугольника трактуется как *визит* во вложенную карту. Среду этой карты будем называть средой визита. Ее особенность в том, что в ней будут определяться дополнительные управляющие атрибуты, используемые для поддержки семантики стабов. Ромбовидная фигура стаба исключена из рисунка, чем акцентируется, что вершина стаба важна на этапе организации дополнительных путей, соединяющих карту верхнего и вложенного уровней, но не существенна на этапе, когда реализуется специфицированное стабом поведение.

Представленный на рисунке простейший случай стаба с одной вложенной картой позволяет увидеть некоторые аналогии между описанием и вызовом процедур в последовательных языках программирования с одной стороны, и описанием и функционированием стабов в языке *UCM*, с другой. При таком сходстве, представленном в табл. 1, отметим, что стабы отличаются значительно более гибкой семантикой, позволяющей синхронизировать процессы, протекающие как вне вершины стаба, так и внутри локальной среды, инициированной процессами, приходящими на входы стаба. Пояснения к этой таблице и перечисленным в ней понятиям семантики языка *UCM* будут следовать из дальнейшего изложения, тогда как сама таблица приведена для того, чтобы наметить аналогии с известными моделями и отличия от них.

Возвращаясь к рис. 1, отметим, что в общем случае у стаба может быть несколько вложенных карт. Рассмотрим еще одну иллюстрацию (рис. 2) к описываемой семантике стабов. Предполагается, что здесь стаб ссылается на две разные вложенные карты, и на каждую из них – со своей схемой коммутации. Графическая картинка теперь отличается тем, что входы и выходы стаба могут соединяться пунктирными линиями с несколькими стартовыми и соответственно конечными вершинами вложенных карт.

Таблица 1. Сопоставление понятий процедуры и стаба

Последовательные языки	***UCM***
Определение процедуры	Вложенная карта (*Plug-in map*)
Вызов процедуры	Стаб
Выполнение вызова процедуры	Переход от стаба к его вложенной карте
Формальные параметры процедуры	Стартовые и конечные вершины путей вложенной карты
Фактические параметры процедуры	Входные и выходные пути стаба
Синхронная передача фактических параметров	Асинхронная передача фактических параметров
Порядок параметров или именованные параметры	Структуры *InBinding* и *OutBinding*
Входные параметры	*InBinding*
Выходные параметры	*OutBinding*
Вызов процедуры по ссылке (и динамическое связывание в объектных языках)	Аналог *PluginBinding*

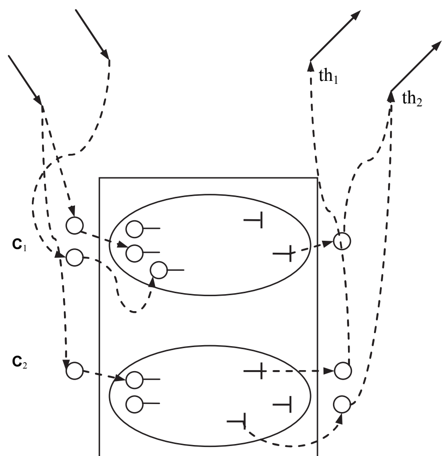

Рис. 2. Схема стаба с двумя вложенными картами

Промежуточная среда визита содержит все вложенные карты стаба, в данном случае – две. На рисунке опущены имена этих карт, но добавлены обозначения C_1, C_2 и th_1, th_2. Первые два – это условия, которые задает автор проекта *UCM* и которые называются предусловиями структур *InBinding*. Смысл предусловий в том, что управление с входа стаба может быть передано на стартовые вершины путей вложенной карты, но только если предусловие истинно. Заметим, что стартовые вершины также могут иметь условия, определяющие старт процессов, но это уже другие условия, связанные не с вложенностью карты, а с ней самой. Выражения th_1 и th_2, значениями которых могут быть целые положительные числа, называются *поро-*

гами (*threshold*) выходных путей стаба[2], которые будут рассмотрены далее. Смысл порогов в том, что фишки могут перейти за пределы стаба только после того, как к пути, отмеченному пороговым выражением, придет столько фишек, какова величина порога. При этом все последующие фишки, приходящие в рамках данного визита к этому пути, будут игнорироваться. Отметим, что даже если выходной путь стаба, отмеченный пороговым выражением, связан только с одной конечной вершиной пути в одной карте, значение этого выражения может быть больше единицы, поскольку одна вложенная карта может сгенерировать несколько фишек. Если условия привязки не указаны, то по умолчанию это трактуется так, как если они равны *true*. Если не указаны пороговые значения выходов стаба, то по умолчанию они приравниваются к числу выходных привязок стаба к этому выходу.

Карты *UCM* могут обладать свойством *singleton*, фиксирующим, что число их возможных экземпляров равно единице. Это свойство, если оно истинно для данной карты, позволяет синхронизировать процессы в разных стабах, обращающихся к общей вложенной карте.

На рис. 3 изображены два стаба, у которых общая вложенная карта. В этом примере процессы внутри карты синхронизируются посредством конструкта *AndJoin* и могут продолжиться

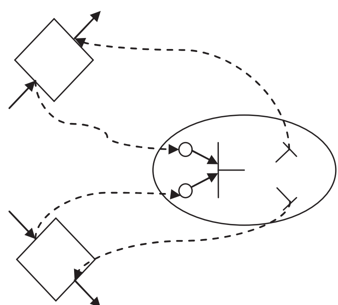

Рис. 3. Использование *singleton* карты

только по достижении обеими такой вершины. Если бы этой вложенной карте не было присуще свойство *singleton*, то процессы каждого стаба продолжались бы в своей копии карты, не смогли бы синхронизироваться и пройти вершину *AndJoin*.

[2] Только для стабов синхронизирующего и блокирующего видов.

4. Параллелизмы, реализуемые в стабах

Язык *UCM* допускает такие параллелизмы процессов, порождаемые стабами:

- между процессами, генерируемыми разными стабами, связанными с одними и теми же вложенными картами;
- между процессами, генерируемыми разными переходами к одному входу того же стаба – это обусловлено возможностью перехода к вложенной карте, не дожидаясь завершения предыдущего перехода;
- между процессами, генерируемыми переходами к разным входам того же стаба.

Для идентификации разных локальных сред и их процессов, порожденных стабом, используется понятие *визита*. К *i*-му визиту относятся процессы во вложенных картах стаба, инициированные *i*-ми фишками на каждом из его входных путей. После создания визита, можно говорить о среде визита и развивающихся в нем процессах. Одновременно могут существовать несколько визитов, в каждом из которых параллельно будут развиваться свои процессы.

5. Виды стабов

На рис. 1 и 2 проиллюстрированы элементы стабов – совокупности вложенных карт, схемы их привязки к входам и выходам стаба, условия привязки карт к входам стаба и пороговые фильтры на его выходах, а также описано понятие визита. В языке *UCM* определяется несколько видов стабов: статические, динамические, синхронизирующие и блокирующие. Графическое их изображение в языке приведено на рис. 4.

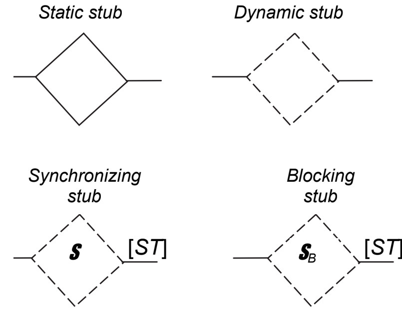

Рис. 4. Виды стабов и их графическое изображение

Только ромб статического стаба прорисовывается непрерывной линией. В синхронизирую-

щем стабе внутри ромба добавляется буква *S*, указывающая на синхронизацию, осуществляемую согласно значению синхронизирующего порога *ST*. Наконец, в блокирующем стабе дополнительно появилась буква *B* как нижний индекс в его спецификации.

Охарактеризуем особенности этих стабов. В случае *статического* стаба число вложенных карт не больше единицы и, если вложенная карта отсутствует, то такой стаб интерпретируется как тупик в процессе. В случае *динамических* стабов нет верхнего ограничения на число вложенных карт. *Синхронизирующий* стаб – это частный случай динамического стаба, но отличается синхронизацией выхода процессов вложенных карт из стаба в рамках среды одного визита. Количество синхронизируемых для выхода процессов задается (явно или по умолчанию) пороговыми выражениями. Выход из синхронизирующего стаба подобен вершине *AndJoin*, поведение которой также специфицируется пороговым выражением. Фишка выходит из синхронизирующего стаба только после накопления определенного числа фишек на выходе. В то же время такой выход отличается от вершин *AndJoin* тем, что в *AndJoin* требовалось получение фишек с каждого входа слияния, а здесь существенно лишь общее количество поступивших фишек, но не те пути, по которым они приходят.

Наконец, *блокирующие* стабы – такой частный случай синхронизирующего стаба, когда в каждый момент времени может выполняться не более одного визита стаба. У таких стабов могут образовываться очереди фишек на их входах, ожидающие завершения текущего визита. Условиями завершения визита будем считать такие:

- по всем входным привязкам уже были инициированы процессы;
- если все конечные вершины вложенной карты достигнуты и все процессы уровня среды визита завершены.

6. Общая схема реализации семантики стабов

Семантика стабов задана правилами 1–16 в табл. 2 для каждого типа стабов группой из четырех правил. Эти четыре группы правил вписываются в одну общую схему, когда все первые (соответственно, все вторые, третьи, четвертые) правила внутри этих групп сходны между собой в том смысле, что они относятся к определенному этапу развития процессов, порождаемых стабом. Содержательно эти этапы можно охарактеризовать как (1) от вершины перед стабом к структурам связывания (*InBinding*) для этого входа, (2) от структур связывания к стартовым вершинам путей вложенных карт, (3) от конечных вершин путей вложенных карт к выходным структурам связывания (*OutBinding*), и (4) от выходных структур связывания к вершине после стаба.

Структура вложенности сред, использованная при реализации семантики стабов, отличается от указанной в стандарте. Это вызвано следующими причинами:

- среда визита не используется для статического и динамического стабов, так как ее свойства проявляются только в случае синхронизирующего (и блокирующего) стаба;
- карты вкладываются в другие карты, но не вкладываются в визиты, а наоборот – визиты вкладываются в карты во избежание дополнительного размножения сред карт.

Таким образом, структуру сред можно представить выражением $E\ [M\ [V\ [C\ [\ P\ ]\ ]]]$, где M, V, C и P – соответственно среды карт, визитов, компонент, а также процессы.

Свойства и управляющие атрибуты для задания семантики стабов. Атрибутные транзиционные системы задаются описанием множества их состояний и переходов. Состояния этих систем описываются совокупностями значений их атрибутов, переходы – правилами, изменяющими значения их атрибутов. В рассматриваемых транзиционных системах атрибуты относятся к конкретным средам – либо к глобальной, либо к средам карт и визитов. Рассматриваются как функциональные атрибуты карт или визитов, и в качестве параметров этих атрибутов используются соответственно имена этих карт и визитов. Атрибуты, значения которых не изменяются никакими правилами переходов, будем называть свойствами соответствующих сред.

В среде карт определяется[3] следующее множество свойств и атрибутов:

• *singleton*: {*MapName*} → *Bool* – свойство, при истинности которого дополнительные экземпляры карты не создаются (см. рис. 3); по умолчанию значение *true* для каждой карты;

• *MapReplica*: {*MapName*} → *Int* – свойство, указывающее число копий карты при формировании визита; по умолчанию значение 1 на всем множестве параметров; для статических стабов значение этого свойства всегда равно единице; если значение больше единицы, то в среду визита погружается столько экземпляров карты, каково для нее значение этого свойства;

• *StubVisit*: {*StubName*} → *Int* – атрибут, указывающий число визитов стаба; начальное значение ноль;

• *StubCounter*: {*StubName*} × {*InPathName*} → *Int* – атрибут, указывающий число фишек, поступивших к вершине *StubName* типа стаб по входному пути *InPathName*; начальное значение атрибута ноль;

• *threshold*:{*StubName*} × {*OutPathName*} → *Int* – атрибут, указывающий пороговое значение выходного пути *OutPathName* для вершины *StubName* типа стаб; если порог выходного пути *OutPathName* не указан, то по умолчанию он вычисляется в момент образования визита как число его вложенных карт;

• *exit*: {*EndPointName*} → {*OutPathName*} – атрибут, реализующий выходные привязки стаба; начальное значение атрибута *Nil* на всей области определения; его значение формируется при погружении карты и происходит это таким образом, что пара из входного параметра и значения атрибута для него в совокупности соответствуют одной из выходных привязок стаба к карте;

• *existvisit*: {*VisitName*} → *Bool* – атрибут, указывающий наличие в среде текущей карты вложенной среды визита с заданным именем; начальное значение атрибута *false*.

Отметим, что атрибут *exit* отличается от остальных тем, что в рамках описания семантики стабов он относится к вложенным картам, тогда как все остальные атрибуты относятся к картам, содержащим стаб.

Атрибуты визитов локализуются в средах, соответствующих именам стабов, из которых осуществляется обращение к ним:

• *visit* – атрибут, указывающий имя визита внутри его среды; для стаба *StubName* начальное значение этого атрибута генерируется при обращении к функции погружения (см. далее правило *insertvisit*), формирующей среду этого визита как текущее значение атрибута карты *StubVisit*(*StubName*);

• *OutPathCounter*: {*OutPathName*} → *Int* – атрибут, указывающий, сколько раз в рамках рассматриваемого визита был достигнут выходной путь *OutPathName* из стаба; начальное значение ноль;

• *enable*: {*OutPathName*} → *Bool* – атрибут, указывающий для каждого выходного пути *OutPathName* стаба, был ли он достигнут в рамках данного визита хотя бы раз; начальное значение *false*;

• *past*: {*OutPathName*} → *Bool* – атрибут, указывающий для каждого выходного пути *OutPathName* стаба, был ли он уже пройден; начальное значение *false*.

Функция погружения. Особую роль в задании правил переходов описываемой транзиционной системы выполняет функция погружения, посредством которой интегрируются изменения в разных компонентах структурированной среды этой системы. В описании семантики стабов используются правила погружения *apply*, *check* и *insert*, описанные в [14], а также представленные здесь правила *insertvisit*, *movein* и *moveout*.

Определяемые далее три новых правила (1–3) связаны с этапами общей схемы реализации стабов. Правило (1) применяется на первом этапе –

[3] Фигурные скобки используются для обозначения множеств объектов, например, {*MapName*} означает множество карт, заданных своими именами.

от входных путей стаба к привязкам *InBinding* (на рис. 1 они представлены как структуры *IB*1 и *IB*2). Правило (2) применяется для моделирования переходов от привязок к стартовым вершинам путей вложенных карт. И наконец, правило (3) применяется при переходах от конечных вершин путей вложенных карт к выходным привязкам (на рис. 1 они представлены как структуры *OB*).

Правило (1), *insertvisit*, используется для динамического формирования сред визитов (девятая и тринадцатая строки в табл. 2):

$$\frac{u \xrightarrow{insertvisit(V)} u'}{M[u \parallel v] \parallel w \xrightarrow{insertvisit(V)} M[V_n[initvisit(V,k);u'] \parallel v] \parallel w}. \quad (1)$$

В этом правиле используются следующие обозначения: V – имя стаба, n – текущее значение атрибута $\mathtt{StubVisit}(V)$, используемое для именования V_n среды формируемого визита, *initvisit* – функция, осуществляющая генерацию начальных значений атрибутов в среде формируемого визита. Параметр k, используемый в приводимом ниже определении функция *initvisit* – это количество выходных привязок (*OutBinding* структур) стаба V к карте M.

$$
\begin{aligned}
& initvisit(V,k) = \; (\\
& \quad visit := StabVisit(V), \\
& \quad StabVisit(V) := StabVisit(V)+1, \\
& \quad for\ K = 1..k(\\
& \qquad enable(out_K(V)) := false, \\
& \qquad past(out_K(V)) := false, \\
& \qquad OutPathCounter(out_K) := 0 \quad) \\
&)
\end{aligned}
$$

В результате в текущей среде создается вложенная среда визита с именем V_n , в которой формируются его локальные атрибуты *visit*, *enable*, *past*, *OutPathCounter*, первый из которых скалярный. У остальных параметром служат вершины, идентифицируемые с выходными путями $out_K(V)$ стаба V, $K = 1..k$, и предназначенные для перехода фишек изнутри этого визита во внешнюю карту, которой принадлежит сам стаб. Заметим, что порождаемая после применения этого правила среда визита вкладывается в карту M.

Правило *movein* представлено в двух вариантах, (2*a*) и (2*b*), разнящихся значением свойства *singleton* погружаемой карты $M2$:

$$\frac{u \xrightarrow{movein(V,M2)} u' \,\&\, \left(singleton(M2)=0\right) \& \,(r = MapReplica(M2))}{M1[u \square v] \square w \xrightarrow{movein(V,M2)} M1[copy(M2_1[initexit(V,M2,n).u')] \square \ldots \square} \times$$

$$\times \frac{1}{copy(M2_r[initexit(V,M2,n).u')] \square v \square w}, \qquad (2a)$$

$$\frac{u \xrightarrow{movein(V,M2)} u' \,\&\, \left(singleton(M2)=1\right) \& \,(r = MapReplica(M2))}{M1[u \square v] \square w \xrightarrow{movein(V,M2)} M1[(M2_1[initexit(V,M2,n).u'] \square \ldots \square} \times$$

$$\times \frac{1}{(M2_r[initexit(V,M2,n).u'] \square v \square w}. \qquad (2b)$$

В обоих случаях параметры перехода $u \xrightarrow{movein(V,M2)} u'$ в посылке правила и контекст применения правила имеют следующую интерпретацию. Вершина u – это входная привязка i-го входного пути стаба V, расположенного внутри карты $M1$, к вложенной карте $M2$, u' – соответствующая стартовая вершина пути карты $M2$. Если обратиться к примеру на рис. 1, то u – это входной путь стаба в одной из структур *IB*1 или *IB*2, а u' – стартовая вершина пути карты, в которую ведет пунктирная стрелка из u. Правило (2*a*) погружает $\mathtt{r}$ копий карты $M2$ в среду карты $M1$, в которой расположен стаб.

Функция *initexit*, формирующая значение атрибута *exit* для карты $M2$ (или ее копий), определяется соотношением

$$initexit(V,M,n) =$$
$$= (for\ K = 1 \ldots n\ (exit(EndPoint_K(M),\ V) := a_K)).$$

В среде каждой такой копии посредством обращения к функции $initexit(V,M,n) = (for\ K = 1 \ldots n\ \ (exit(EndPoint_K(M),\ V) := a_K))$ формируется такое значение функционального атрибута *exit*, когда каждой конечной вершине *EndPoint_K* карты M сопоставляется выходная вершина стаба V, образующая вместе с ней выходную привязку стаба (в обозначениях рис. 1 и 2 – это структуры *OB*, ассоциированные с картой $M2$). И также в среду каждой такой копии карты $M2$ погружается u', стартовая вершина процесса в $M2$. Правило (2*b*) отличается от правила (2*a*) истинностью свойства *singleton* для карты $M2$. Если это свойство карты $M2$ истинно, то погружается сама карта, а не ее копии.

Правило (3), *moveout*, определяет переход фишки из вложенной карты в среду карты верхнего уровня:

$$\frac{u \xrightarrow{moveout(exit)} \Delta}{M1[M2[u \parallel v] \parallel w] \parallel z \xrightarrow{moveout(exit)} M1[M2[v] \parallel exit \parallel w] \parallel z}. \quad (3)$$

Контекст этого перехода $u \xrightarrow{moveout(exit)} \Delta$ следующий: u – это некоторая конечная вершина пути, пусть для определенности *ep*, вложенной карты *M*2, *exit* – соответствующая *ep* выходная привязка, определяемая как значение *exit*(*ep*) функционального атрибута, Δ – символ завершения процесса внутри вложенной карты. В результате применения этого правила фишка перемещается из вложенной карты к соответствующей точке выходной привязки (в терминах рис. 1 это некая структура *OB*, ассоциированная с картой *M*2), от которой процесс ее перемещений может продолжиться.

7. Уравнения, реализующие процессы в стабах

Далее приводится табл. 2, в которой даны уравнения четырех видов в алгебре поведений, определяющие семантику стабов: уравнения (1)–(4) для статических стабов, уравнения (5)–(8) для динамических стабов, уравнения (9)–(12) – для синхронизирующих динамических стабов, и (13)–(16) – для блокирующих синхронизирующих динамических стабов. Для краткости в столбце «Конструкт» этой таблицы будет использована только основная часть названия каждого типа, т.е. статический, динамический, синхронизирующий или блокирующий.

Уравнения (1)–(8), определяющие семантику статических и динамических стабов, достаточно просты, их легко можно проиллюстрировать (см. рис. 1 и 2), поэтому подробные пояснения к ним здесь опускаются. Уравнения для синхронизирующих и блокирующих стабов почти идентичны: в частности (14), (15) и (16) соответственно повторяют уравнения (10), (11) и (12). Уравнение (13) отличается от уравнения (9) тем, что в нем дополнительно учитывается атрибут `existvisit`, посредством которого блокируется создание новых визитов для заданного стаба. Поэтому после представления табл. 2 будут даны достаточно детальные комментарии к правилам 9–12 и к отличиям правила 13 от правила 9.

Таблица 2. Уравнения в алгебре поведений, определяющие семантику стабов

№	Конструкт	*UCM* представление	Уравнение	Комментарий
1	2	3	4	5
1	К входной привязке ***статического*** стаба	S in T out1 outn	`S = T_in`	*T* – стаб, *in* – входной путь *T* от вершины *S*, *T_in* – входная привязка
2	От входной привязки статического стаба к вложенной карте	S_in T	`S_in = movein(S,m).T`	*S_in* – входная привязка стаба *S*, *m* – вложенная карта со стартовой вершиной *T*
3	От вложенной карты к выходной привязке статического стаба	S c T_out	`S = check(c).` `moveout(exit(out))`	*S* – конечная вершина вложенной карты, *c* – условие завершения *S*, *T* – стаб, *T_out* – его выходная привязка, соответствующая *S*
4	От выходной привязки статического стаба	S_out T	`S_out = T`	*S* – стаб, *out* – выходная привязка *S*
5	К входной привязке ***динамического*** стаба	S in T out1 outn	`S = insert(T_in_m1, …, T_in_mp)`	*T* – стаб, *in* – входной путь *T*, *m*1, …, *mp* – вложенные карты стаба

1	2	3	4	5
6	От входной привязки динамического стаба к вложенной карте		*S_in_m = check(c).movein(S,m).T*	*S* – стаб, *in* – входной путь *S*, *m* – вложенная карта, *c* – условия привязки карты *m*, *T* – стартовая вершина в m
7	От вложенной карты к выходной привязке динамического стаба		*S = check(c).moveout(exit(out))*	*S* – конечная вершина вложенной карты, *c* – условие завершения *S*, *T* – стаб, *T_out* – выходная привязка *T*, соответствующая *S*
8	От выходной привязки динамического стаба		*S_out = T*	*S* – стаб, *T* – вершина, следующая за *S* на выходном пути *out*, *S_out* – выходная привязка стаба
9	К входной привязке ***синхронизирующего*** стаба		*S =* *check(StubCounter(T, in) = StubVisit(T)).* *apply(StubCounter(T,* in) *:= StubCounter(T,in)*+1, *StubVisit(T):=StubVisit(T)* + 1)*.insertvisit(T).* *(insert(check(visit = StubCounter(T,in1)).(T_in1_m1, …, T_in1_mp))* || … || *insert(check(visit =* StubCounter(*T*, ink)).(T_ink_m1, …, T_ink_mp))) + *(check(StubCounter(T, in) < StubVisit(T)).apply(StubCounter(T, in) :=* *StubCounter(T, in)* + 1))	*T* – стаб с вершиной *S*, предшествующей ему на входном пути *in* *m*1, …, *mp* – вложенные карты, *n* – количество выходных путей *T*, *in*1, …, *ink* – входные пути *T*, *T_inq_mr* – входная привязка пути *inq* для карты *mr*
10	От входной привязки синхронизирующего стаба к вложенной карте		*S_in_m = check(c).movein(S,m).T*	*S* – стаб с входной привязкой *in*, *m* – вложенная карта, *c* – условие привязки, *T* – стартовая вершина в карте *m*
11	От вложенной карты к выходной привязке синхронизирующего стаба		*S = check(c & (enable(out) = false) & (past(out) = false)).apply(enable(out) := true, OutPathCounter(out) := OutPathCounter(out) +* 1)*.moveout(exit(out)) + check(c & (enable(out) = true) & (past(out)= false)).apply(OutPathCounter(out) := OutPathCounter(out) +* 1) *+ check(c & (past(out) = true))*	*S* – конечная вершина пути карты плагина, *c* – условие завершения *S*, *T* – синхронизирующий стаб, *out* – выходная привязка *T*, соответствующая *S*
12	От выходной привязки синхронизирующего стаба		*S_out = check(OutPathCounter(out) >= threshold(S, out)).apply(past(out) := true).T*	*S* – стаб, *out* – выходной путь из *S* в *T* , *S_out* – выходная привязка
13	К входной привязке ***блокирующего*** стаба		*S = check((StubCounter(T,in) <= StubVisit*(T)) *& ~existvisit(T_1)).apply(StubCounter(T,in):= StubCounter(T,in)+1, StubVisit*(T)*:=StubVisit(T)*+1)*.* *insertvisit(T).* *(insert(check(StubCounter(T, in*1) > 0)*.(T_in1_m1, …, T_in1_mp))* || … || *insert(check(StubCounter(T, ink)* > 0)*.(T_ink_m1, …, T_ink_mp)))* + *check((StubCounter(T, in) < StubVisit(T)) & existvisit(T_1)).apply(StubCounter(T,in):=StubCounter(T, in)* + 1)	*T* – блокирующий стаб, *in* – входной путь *T* из *S*, *m*1, …, *mp* – вложенные карты, *n* – количество выходных путей *T*, *in*1, …, *ink* – входные пути *T*

Продолжение табл. 2

14	От входной привязки блокирующего стаба к вложенной карте		S_in_m = check(c).movein(S,m).T	*S* – блокирующий стаб с входным путем *in*, *m* – вложенная карта, *c* – условия привязки для *m*, *T* – стартовая вершина пути карты *m*
15	От вложенной карты к выходной привязке блокирующего стаба		S = check(c & (enable(out) = false) & (past(out) = false)).apply(enable(out) := true, OutPathCounter(out) := OutPathCounter(out) + 1).moveout(exit(out)) + check(c & (enable(out) = true) & (past(out) = false)).apply(OutPathCounter(out) := OutPathCounter(out) + 1) + check(c & (past(out) = true))	*S* – конечная вершина пути вложенной карты, *c* – условие *S*, *T* – блокирующий стаб, *out* – выходной путь *T*, *T_out* – выходная привязка, соответствующая *S*
16	От выходной привязки блокирующего стаба		S_out = check(OutPathCounter(out) = threshold(S, out)).apply(past(out) := true).T	*S* – блокирующий стаб, *out* – выходной путь из *S* в *T* , *S_out* выходная привязка

Комментарий к правилу 9. Правило содержит две альтернативы, разделенные знаком +. Одна – описывает случай StubCounter(T,in)= StubVisit(T), когда для фишки, пришедшей к стабу *T* по входу in должен быть создан новый визит. Другая соответствует случаю StubCounter(T,in) < StubVisit(T), когда пришедшая фишка относится к одному из уже созданных визитов. Действие, выполняемое во втором случае, сводится к увеличению счетчика StubCounter(T, in) на единицу. В свою очередь, результат этого действия может сделать истинным равенство visit = StubCounter(T, in) внутри одного из ранее созданных визитов, что позволит осуществлять переходы для других фишек внутри визита.

Выполнение равенства StubCounter(T, in)=StubVisit(T) в первой альтернативе влечет следующую последовательность действий:

- увеличение обоих счетчиков StubCounter(T, in) и StubVisit(T);
- создание нового визита обращением к правилу погружения insertvisit(T);
- погружение в среду этого визита всех входных привязок T_in1_m1,..., T_ink_mp для всех карт этого стаба.

Последнее действие представляет собой параллельное погружение агентов, соответствующих k входам стаба. Рассмотрим ветвь для входа in1. Она начинается с проверки уже упомянутого равенства visit = StubCounter(T, in1), за которым следует копирование фишки в p параллельных ветвях, по одной для каждой карты m1,..., mp.

Комментарий к правилу 10. Это правило предполагает два действия: проверку истинности предусловия структуры *PluginBinding* для карты m и ее погружение в среду визита.

Комментарий к правилу 11. Этим правилом специфицируется передача фишки от конечных вершин путей вложенных карт к выходным привязкам. Уравнение перехода содержит три альтернативы.

Первая альтернатива, когда переход фишки к привязке не состоялся, описывается поведением check(c & enable(out)=false) & &(past(out)=false)).apply(enable(out):= := true, OutPathCounter(out):=OutPathCounter(out)+1).moveout(exit(out)). После того, как выполнена эта альтернатива, т.е. проверено, что переход может состояться, счетчик числа перешедших фишек увеличивается на единицу и посредством функции moveout фишка переходит к выходной привязке. Вторая альтернатива с поведением check(c & & (enable(out) = true) & (past(out) =

`false)).apply(OutPathCounter(out):= := OutPathCounter(out)+1)` соответствует случаю, когда уже имеются фишки для данной выходной привязки, достигшие конечных вершин путей, но их число еще меньше порогового значения, что не позволяет фишке перейти к выходной привязке. После проверки условия этой альтернативы увеличивается на единицу счетчик выходного пути, но фишка к выходной привязке не переходит, поскольку она уже перешла туда по первой альтернативе. Наконец, третья альтернатива `check(c & & (past(out) = true))` представляет собой случай, когда число достигнутых конечных вершин путей для данной выходной привязки уже больше или равно пороговому значению, и фишка перешла из выходной привязки стаба по пути `out` к следующей за стабом вершине пути.

Комментарий к правилу 12. Это правило `S_out = check(OutPathCounter(out)>= threshold(S,out)).apply(past(out):= := true).T` описывает переход фишки от выходной привязки *S_out* к вершине *T*, непосредственно следующей за стабом *S* на пути *out*.

Комментарий к правилу 13. Это правило отличается от правила 9 тем, что условия выполнения каждой из двух его альтернатив дополнительно предполагают проверку истинностного значения `existvisit(T_1)`. Здесь первая поведенческая альтернатива в уравнении позволяет формировать новый визит для блокирующего стаба *T* только если в среде карты, которой принадлежит стаб, не существует других визитов для этого стаба. Вторая альтернатива описывает обработку фишки текущего визита, которая пришла по входу `in`.

8. Заключение. В работе применен инсерционный подход для формализованного описания семантики стабов в языке параллельного моделирования *UCM*. Для каждого из представленных видов стабов (статический, динамический, синхронизирующий и блокирующий) приведено описание семантики в алгебре процессов, использующее функцию погружения и систему управляющих атрибутов. В то же время, учитывая все многообразие средств языка *UCM* и ситуаций, возможных при развитии процессов, специфицируемых в нем, некоторые детали в описании семантики опущены. Например, не представлены правила, применяемые при удалении сред.

Рассматриваемая семантика охватывает различные известные виды параллелизма: *параллелизм между заданиями*, который проявляется как параллелизм между обращениями к одной и той же карте из разных стабов; *параллелизм по данным*, поступающими на разные входы; *потоковый*, или *pipeline параллелизм*, который наблюдается между разными (по времени) обращениями к одному и тому же стабу.

Стаб – это, наряду с понятием *компонент*, тот конструкт языка *UCM*, который привносит в него иерархию сред. Предпринята попытка увидеть аналогии между стабами *UCM* и процедурами в языках программирования, которые служат испытанным средством для уменьшения повторов в программных текстах. По мнению автора, в стабах можно было бы в полной мере имплементировать и процедурную парадигму, причем сохраняя все многообразие их видов. Достаточно было бы вынести описание стабов в отдельный раздел, провести их именование, а также упорядочение на множестве входов и выходов.

Автор выражает признательность А.А. Губе и К.И. Шушпанову за обсуждения по семантике языка *UCM*.

1. *International* Telecommunications Union. Recommendation Z.120 – Message Sequence Charts (MSC), 1998.
2. *Ibid*. Recommendation Z.100 – Specification and Description Language (SDL), 1999. – 232 p. – http://www.itu.int/ITU-T/studygroups/com10/languages/Z.1001199.pdf
3. *Object* Management Group. Unified Modeling Language Specification, 2.0. – 2003. – 624 p. – http;//www.sparxsystems.com.au/bin/UML2SuperStructure.pdf
4. *International* Telecommunications Union. Recommendation Z.151 – User Requirements Notation (URN), 2008. – 190 p. – http://www.itu.int/rec/T-REC-Z.151-200811-I/en
5. *Инсерционное* программирование / А.А. Летичевский, Ю.В. Капитонова, В.А. Волков и др. // Кибернетика и системный анализ. – 2003. – № 1. – С. 12–32.
6. *Milner R*. Communication and Concurrency // Int. Series in Comp. Sci. – Prentice-Hall, 1989. – 300 p.
7. *Hoare C.A.R*. Communicating Sequential Processes // Ibid, 1985. – http://www.usingcsp.com/cspbook.pdf

8. *Kealey J.*, *Amyot D*. Enhanced Use Case Map Traversal Semantics / 13th SDL Forum (SDL'07). – Paris, France: Springer, LNCS 4745. – P. 133–149. (Sept. 2007). – http://dl.acm.org/citation.cfm?id=1779946
9. *Mussbacher G*. Aspect-Oriented User Requirements Notation / Thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of Ph.D. in Comp. Sci. University of Ottawa, Ottawa, Canada, Sept. 2010. – P. 326. – http://lotos.sci.uottava.ca/ucm/pub/UCM/VirLibGunterPhDThesis/AspectOriented_User_Requirements_Notation.pdf
10. *Hassine J*., *Rilling J*., *Dssouli R*. An ASM Operational Semantics for Use Case Maps / 13th IEEE Int. Requirement Eng. Conf. (RE'05), IEEE Computer Society Press, Sept. 2005. – P. 467–468. – http://dl.acm.org/citation.cfm/?id=1100666
11. *Jameleddine Hassine*. Formal semantics and verification of use case maps / A thesis in The Depart. of Comp. Sci. and Software Eng., Concordia University, Montreal, Quebec, Canada, 2008. – 284 p. – http://lotos.site.uottava.ca/ucm/pub/UCM/VirLibJamelPhDThesis/phd_ih.pdf
12. *Weiss M*., *Amyot D*. Business Process Modeling with URN // Int. J. of E-Business Research. – July-Sept. 2005. – N 3, **1**. – P. 63–90.
13. *ISO*/*IEC*. Software and system engineering. High-level Petri nets, part 1, Concepts, definitions and graphical notation, 2004. – 38 p. – http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38225
14. *Губа А.А*., *Шушпанов К.И*. Инсерционная семантика плоских многопотоковых моделей языка *UCM* // УСиМ – № 6. – С. 15–21, 34.

Тел. для справок: +38 067 468-6035 (Киев)
© А.Б. Годлевский, 2012

●