УДК 519.7+004.436.4

А.А. Губа, К.И. Шушпанов

Инсерционная семантика плоских многопотоковых моделей языка *UCM*

Предложена математическая семантика плоских многопотоковых моделей языка *UCM*. В качестве формализма используется инсерционная модель, обогащенная взаимодействием параллельных процессов через общую память. Разработана функция погружения параллельных процессов *UCM* в среду. Семантика предложена для символьной верификации многопотоковых моделей.

The mathematical semantics of flattened multithreaded models of UCM language is suggested. The insertional model enriched by the interaction of parallel processes via shared memory is used as a formalism. An insertion function for the parallel UCM processes is developed. The semantics is suggested for the symbolic verification of multithreaded models.

Запропоновано математичну семантику пласких багатопотокових моделей мови *UCM*. Як формалізм використано інсерційну модель, збагачену взаємодією паралельних процесів через загальну пам'ять. Розроблено функцію занурення паралельних процесів *UCM* в середовище. Семантику запропоновано для символьної верифікації багатопотокових моделей.

Введение. Последние десятилетия активно развивались программы для формализации требований, их поддержки и повторного использования. Одной из последних разработок в этой сфере стало создание языка *User Requirements Notation* (*URN*, или нотации требований пользователя) [1]. *URN* – язык высокого уровня, который помогает разрабатывать и изучать модели систем сложной структуры и поведения; связывает поведение системы с ее архитектурой в формальной и визуальной форме. Основные области применения – это телекоммуникационные системы, сервисы, бизнес-процессы, но чаще *URN* применяется для описания реактивных и многопотоковых систем. Состоит из нефункциональных (*URN–NFR*) и функциональных (*URN–FR*) требований.

Подмножество языка *URN–NFR* называется *Goal-oriented Requirement Language* (*GRL*) и служит для поддержки целеориентированного моделирования, вводит конструкции для выражения понятий разных типов, которые появляются при разработке требований. Подмножество языка *URN–FR* называется *Use Case Map* (*UCM*). Спецификации *UCM* используют пути сценариев для иллюстрации причинно-следственных связей между действиями системы в ее работе. Кроме того, *UCM* обеспечивают комплексное представление о поведении и структуре системы, позволяя накладывать пути сценариев на структуру абстрактных компонент. Сочетание поведения и структуры позволяет выражать архитектуру системы, после чего возможно уточнение до более детальных сценарных моделей, таких, как диаграммы *MSC* [2], диаграммы последовательностей или диаграммы состояний *UML* [3], конечные автоматы *SDL* [4] и, наконец, до кода конкретной реализации. Таким образом, в *UCM* возможны различные уровни абстракции.

Семантику *UCM* начали разрабатывать в начале 1990-х годов. Первой попыткой построения семантики *UCM* было введение механизма обхода *UCM* [5, 6], который позволяет выделить конкретный сценарий в модели на языке *UCM* и преобразовать его в *MSC*. Позже семантика *UCM* выражалась формализмами языка *LOTOS* [7], абстрактных машин состояний (*ASM*) [8], а также временных транзиционных систем (*CTS*) и временных автоматов (*TA*) [9]. Но интерпретация моделей на языке *UCM* этими формализмами не поддерживала таких задач, как инстанциализация карт *UCM*, привязка к компонентам и картам. Работы по семан-

Ключевые слова: *UCM*, символьная верификация, инсерционное моделирование.

тике *UCM* основаны на моделировании бизнес-процессов [10–12] и используют теорию сетей Петри [13, 14]. Механизм обхода модели на языке *UCM*, основанный на такой семантике *UCM*, работает только с конкретными значениями и заданными пользователем сценариями. Однако символьная верификация модели без конкретных значений для любых возможных сценариев остается актуальной задачей. В данной статье формализмом для описания семантики *UCM* стала инсерционная модель [15], что позволяет использовать дедуктивные средства символьного моделирования для верификации и генерации трасс в сочетании с моделированием поведения системы методом проверки моделей.

Спецификация *UCM*

Данный проект представляет собой совокупность карт *UCM*, состоящих из конечного (возможно пустого) множества путей *UCM* (*Paths*). Карты есть базовым понятием языка и используются для задания поведенческих сценариев транзиционных систем. Это графовые структуры, вершины которых ассоциированы с языковыми конструктами.

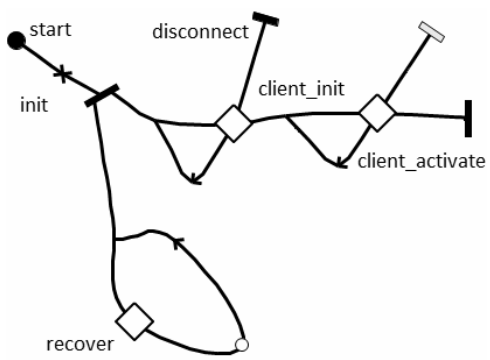


Рис. 1. Пример карты *UCM*

Они содержат вершины, расположенные на путях, пролегающих внутри одной карты. Пути определяют порядок следования вершин. Последние могут быть конструктами разных типов, используемых для начала и завершения путей, их альтернативных и параллельных ветвлений и слияний. Пути, расположенные на разных картах, могут связываться между собой вершинами типа *стаб* [16]. Вершины могут находиться внутри компонент, которые есть абстрактными сущностями, используемыми для структуризации модели, и могут быть вложенными. В *UCM* могут быть определены переменные, используемые в условиях и действиях, указанных в вершинах путей.

В данной статье описаны конструкты для плоских многопотоковых моделей на языке *UCM*. Плоские модели не содержат конструктов компонент и стабов, которые представляют собой средства структурирования моделей. Многопотоковые системы – это системы параллельно взаимодействующих процессов, в которых взаимодействие осуществляется через общую память. Примерами таких систем могут быть программы для многоядерных процессоров. Многопотоковость в *UCM* реализуется параллельными путями и их параллельными ветвлениями.

Язык *UCM* позволяет легко развивать проект, пополняя его, например, новыми картами и путями. Такой способ инкрементального проектирования системы предоставляет богатые возможности разработчикам, но, в то же время, чреват возникновением коллизий уже на самом верхнем уровне описания системы. В связи с этим актуальна верификация проектов *UCM* с целью раннего выявления противоречий. В частности, предполагается рассматривать такие задачи, как достижимость, обнаружение тупиковых состояний и проверка свойств безопасности. Решение задач верификации проектов *UCM* предполагает наличие математической семантики языка *UCM*, которая позволила бы символьно моделировать процессы, допустимые в рамках исследуемой спецификации системы. Предлагаемая авторами формальная семантика языка *UCM* основывается на инсерционной модели этого языка.

Инсерционная модель

Модель строится по проекту *UCM*, создаваемому разработчиком, и включает в себя следующие составляющие:

- описание среды `E`, состоящей из агентов системы `u1`, `u2`, `...`, `un` и определения атрибутов;
- систему уравнений, описывающую поведение агентов;

• функцию погружения, описывающую изменение состояния среды E[u1, u2, ..., un] и погруженных в нее агентов.

Среда, описывающая конкретный проект *UCM* – это среда верхнего уровня для вложенных в нее процессов как агентов нижнего уровня, взаимодействующих между собой через атрибуты в общей памяти.

Все атрибуты среды делятся на два непересекающихся множества: пользовательские и управляющие. Пользовательские атрибуты соответствуют переменным проекта *UCM* (*Variables*), которые вводятся разработчиком. Они могут быть следующих типов: логического (*bool*), целого (*int*), перечислимого (*enum*). Их значение может проверяться в условиях (*Conditions*) и изменяться в действиях (*Responsibilities*). Управляющие атрибуты создаются на этапе построения модели проекта *UCM* (трансляции) в систему сред и агентов с целью обеспечить соответствующее семантике *UCM* поведение процессов.

Любой процесс есть либо действие, либо композиция процессов, образованной с помощью операций префиксинга, недетерминированного выбора и параллельной композиции с семантикой интерливинга. Начальное состояние среды – это параллельная композиция стартовых точек (*StartPoints*), рассматриваемых как начальные состояния агентов, погруженных в среду карт, а также как множество начальных значений атрибутов.

Переходы в системе определяются уравнениями вида:

$S = a.S_1$; $S = S_1 + S_2$; $S = S_1 \,||\, S_2$,

в которых S – текущее состояние агента (процесса), погруженного в среду. В качестве S рассматривается вершина пути и ассоциированный с ней конструкт языка *UCM*. Первое уравнение описывает переход агента из состояния S в состояние S_1 после выполнения действия, задаваемого префиксом a. Состояние S_1 в первом уравнении может отсутствовать. Такой переход интерпретируется как успешное завершение агента и удаление его из среды. Второе уравнение задает альтернативные переходы агента в состояние S_1 или S_2. В третьем случае вместо одного агента в состоянии S в среду погружаются два агента в состояниях S_1 и S_2 соответственно.

Возможны два вида инсерционных моделей – конкретные и символьные. Состояние конкретной модели описывается конкретными значениями ее атрибутов из соответствующих им доменов. Состояние символьной модели представляется формулой, определяющей множество допустимых значений атрибутов.

В предлагаемой далее семантике конструктов *UCM* управляющие атрибуты конкретны, а пользовательские атрибуты – символьные.

Функция погружения описывает взаимодействие между агентами и средой, в которую они погружаются или уже погружены. Поскольку правила, определяющие функцию погружения, используют конкретные управляющие атрибуты, математическое описание этих правил в данной работе будет следовать за описанием атрибутов. Краткая характеристика правил, определяющих функцию погружения, будет даваться в виде перечисления ситуаций, когда применяются эти правила. Такие ситуации связаны с процедурами:

• выполнения действия;

• вычисления условия, определяющего дальнейшее развитие процесса;

• образования нового процесса, т.е. погружения его начальной вершины в среду.

Отличие погруженных в среду агентов от остальных состоит в том, что заданные уравнениями правила переходов могут применяться только к уже погруженным агентам.

Семантика плоских моделей на языке *UCM*

В этом разделе рассмотрим семантику следующих конструктов *UCM* – вершин пути:

• *StartPoint* (начало пути),

• *EndPoint* (конец пути),

• *EmptyPoint* (пустой конструкт),

• *DirectionArrow* (направление пути),

• *Responsibility* (действие),

• *OrFork* (альтернативное ветвление),

• *OrJoin* (соединение альтернатив без синхронизации),

• *AndFork* (параллельное ветвление),

• *AndJoin* (соединение альтернатив с синхронизацией),

• *WaitingPlace* (ожидание),

• *Timer* (таймер).

Агенты могут совершать следующие действия:

• *apply*(r, a),

• *check*(c),

• *insert*(v).

Описание семантики этих действий определяется следующими правилами:

Правило apply(r, a) имеет вид

$$\frac{E \xrightarrow{apply(r,a)} E', u \xrightarrow{apply(r,a)} u'}{E[u] \xrightarrow{apply(r,a)} E'[u']},$$

Рис. 2. Правило *apply*

где r – количество раз выполнения этого действия; a – некоторое действие над атрибутами; E, E′ – состояния среды; u, u′ – состояния агента. По данному правилу применяется действие к текущему состоянию среды.

Правило *check*(c) имеет вид

$$\frac{u \xrightarrow{check(c)} u', E \wedge c \neq 0}{E[u] \xrightarrow{check(c)} E[u']},$$

Рис. 3. Правило *check*

где c – некоторое условие над атрибутами. Правило проверяет истинность условия, и если оно истинно – процесс продолжается, а если нет – процесс ожидает его истинности.

Правило *insert*(v) имеет вид

$$\frac{u \xrightarrow{insert(v)} u'}{E[u] \xrightarrow{insert(v)} E[v \,||\, u']},$$

Рис. 4. Правило *insert*

где v – новый процесс. Правило погружает новый процесс в среду.

В табл. 1–5 описана поведенческая семантика конструктов *UCM*. Выражения в столбцах означают:

• порядковый номер правила;

• описание использования конструкта;

• графическое изображение использования конструкта;

• уравнение, интерпретирующее использование конструкта;

• комментарий для обозначений в третьем и четвертом столбцах.

В таблицах текущее состояние обозначается как S (*Source*), следующее – как T (Target).

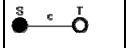
Конструкты Responsibility и StartPoint

Конструкт *Responsibility* использует следующий атрибут:

ResponsibilityReplica: (*ResponsibilityName*) → *int*,

где *ResponsibilityName* – множество всех имен конструктов *Responsibility*. Этот атрибут используется в правиле *apply* для указания количества раз выполнения действия. Значение атрибута задается при создании проекта и остается неизменным. По умолчанию значение *ResponsibilityReplica* равно единице на всей области параметров.

Таблица 1. Семантика конструктов *Responsibility* и *StartPoint*

№	Описание	Изображение	Уравнение	Комментарий
1	Действие		S = *apply* (*Responsibility Replica*(S), a).T	S – *Responsibility* a – действие
2	Начало пути		S = = *check*(c).T	S – *StartPoint* c – условие

Конструкт EndPoint

Конструкт *EndPoint* может использоваться для синхронного соединения двух путей. В этом случае он связан с началом пути, ожиданием и таймером. Конструкты *ожидание* и *таймер* могут быть транзитивными (*transient*) и персистентными (*persistent*), что определяется наличием соответствующего атрибута:

TransientTrigger: (*TransientWaitingPlaceOrTimerName*) → *bool*,

PersistentTrigger: (*PersistentWaitingPlaceOrTimerName*) → *int*,

где *TransientWaitingPlaceOrTimerName* и *PersistentWaitingPlaceOrTi-*

Таблица 2. Семантика конструкта *EndPoint*

№	Описание	Изображение	Уравнение	Комментарий
3	Конец пути		S = *check*(c)	S – *EndPoint* c – условие
4	Конец пути, связанный с началом другого пути		S = *check*(c).*insert*(T)	S – *EndPoint*, T – StartPoint c – условие
5	Конец пути, связанный с транзитивным ожиданием		S = *check*(c).*apply*(*TransientTrigger*(T) := *true*)	S – EndPoint, T – WaitingPlace c – условие
6	Конец пути, связанный с персистентным ожиданием		S = *check*(c).*apply*(*PersistentTrigger*(T) := *PersistentTrigger*(T) + 1)	S – EndPoint, T – WaitingPlace c – условие
7	Конец пути, связанный с транзитивным таймером без пути исчерпания		S = *check*(c).*apply*(*TransientTrigger*(T) := *true*)	S – EndPoint, T – Timer c – условие
8	Конец пути, связанный с персистентным таймером без пути исчерпания		S = *check*(c).*apply*(*PersistentTrigger*(T) := *PersistentTrigger*(T) + 1)	S – EndPoint, T – Timer c – условие
9	Конец пути, связанный с транзитивным таймером с путем исчерпания		S = *check*(c).*apply*(*TransientTrigger*(T) := *true*)	S – EndPoint, T – Timer c – условие
10	Конец пути, связанный с персистентным таймером с путем исчерпания		S = *check*(c).*apply*(*PersistentTrigger*(T) := *PersistentTrigger*(T) + 1)	S – EndPoint, T – Timer c – условие

merName – множество всех имен конструктов *WaitingPlace* и *Timer*. Атрибут *TransientTrigger* указывает достижение транзитивного ожидания или таймера. Начальное значение атрибута равно *false* на всей области параметров. Атрибут *PersistentTrigger* указывает количество достижений персистентного ожидания или таймера. Начальное значение атрибута равно нулю на всей области параметров.

Конструкты OrFork, OrJoin, AndFork и AndJoin

Конструкт *AndJoin* использует атрибуты:

Таблица 3. Семантика конструктов *OrFork*, *OrJoin*, *AndFork* и *AndJoin*

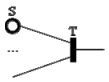
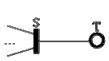
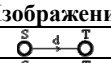
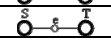
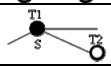
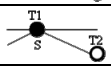
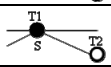
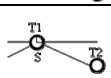
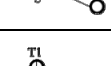
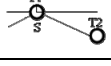
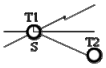
№	Описание	Изображение	Уравнение	Комментарий
11	Альтернативное ветвление		*S* = *check*(*c*1).*T*1 + … + *check*(*cn*).*Tn*	S – *OrFork*, c1, …, cn – условия
12	Соединение альтернатив без синхронизации		*S* = *T*	S – OrJoin
13	Параллельное ветвление		*S* = *insert*(*T*1, …, *Tn*)	S – *AndFork*
14	Переход к соединению альтернатив с синхронизацией		*S* = *check*(*exist*(*T*) = *false*).*apply*(*exist*(*T*) := *true*, *AndJoinCounter*(T, *n*) := *AndJoinCounter*(T, *n*) + 1).*insert*(*T*) + *check*(*exist*(*T*) = *true*).*apply*(*AndJoinCounter*(*T*, *n*) := *AndJoinCounter*(*T*, *n*) + 1)	T – *AndJoin*, n – номер перехода S–T
15	Переход от соединения альтернатив с синхронизацией		*S* = *check*((*AndJoinCounter*(*S*, 1) > 0) & … …& (*AndJoinCounter*(*S*, *n*) > 0)).*apply*(*exist*(*S*) := *false*, *AndJoinCounter*(*S*, 1) := *AndJoinCounter*(*S*, 1) – 1, …, *AndJoinCounter*(*S*, *n*) := *AndJoinCounter*(*S*, *n*) – 1).*T*	S – *AndJoin*

Таблица 4. Семантика конструктов *DirectionArrow* и *EmptyPoint*

№	Описание	Изображение	Уравнение	Комментарий
16	Направление пути		*S = T*	*d – DirectionArrow*
17	Пустой конструкт		*S = T*	*E – EmptyPoint*
18	Пустой конструкт, связанный с началом пути		*S = insert*(*T1*).*T2*	*S – EmptyPoint*, *T1 – StartPoint*
19	Пустой конструкт, связанный с транзитивным ожиданием		*S = apply*(*TransientTrigger*(*T1*) := *true*).*T2*	*S – EmptyPoint*, *T1 – WaitingPlace*
20	Пустой конструкт, связанный с персистентным ожиданием		*S = apply*(*PersistentTrigger*(*T1*) := *PersistentTrigger*(*T1*) + 1).*T2*	*S – EmptyPoint*, *T1 – WaitingPlace*
21	Пустой конструкт, связанный с транзитивным таймером без пути исчерпания		*S = apply*(*TransientTrigger*(*T1*) := *true*).*T2*	*S – EmptyPoint*, *T1 – Timer*
22	Пустой конструкт, связанный с персистентным таймером без пути исчерпания		*S = apply*(*PersistentTrigger*(*T1*) := *PersistentTrigger*(*T1*) + 1).*T2*	*S – EmptyPoint*, *T1 – Timer*
23	Пустой конструкт, связанный с транзитивным таймером с путем исчерпания		*S = apply*(*TransientTrigger*(*T1*) := *true*).*T2*	*S – EmptyPoint*, *T1 – Timer*
24	Пустой конструкт, связанный с персистентным таймером с путем исчерпания		*S = apply*(*PersistentTrigger*(*T1*) := *PersistentTrigger*(*T1*) + 1).*T2*	*S – EmptyPoint*, *T1 – Timer*

exist: (*AndJoinName*) → *bool*,
AndJoinCounter: (*AndJoinName*, *InputNumber*) → *int*,

где *AndJoinName* – множество всех имен конструктов *AndJoin*, *InputNumber* – номер входящего в *AndJoin* пути. Атрибут *exist* указывает достижение соединения альтернатив с синхронизацией. Начальное значение атрибута равно *false* на всей области параметров. Атрибут *AndJoinCounter* указывает количество достижений соединения альтернатив с синхронизацией по входящему пути с номером *InputNumber*. Начальное значение атрибута равно нулю на всей области параметров.

Конструкты DirectionArrow и EmptyPoint

Конструкт *EmptyPoint* может использоваться для асинхронного соединения двух путей. В этом случае он связан с началом пути, ожиданием и таймером.

Конструкты WaitingPlace и Timer

Конструкт Timer использует атрибут

timer: (*TimerName*) → *int*,

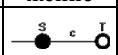
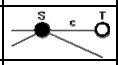
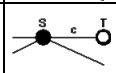
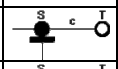
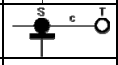
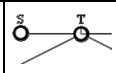
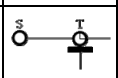
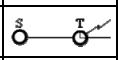
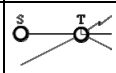
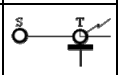
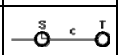
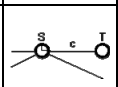
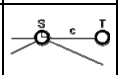
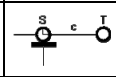
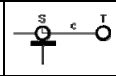
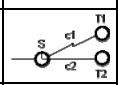
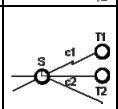
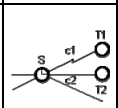
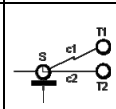
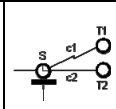
где *TimerName* – множество всех имен конструктов *Timer*. Атрибут *timer* указывает количество установок таймера. Начальное значение атрибута равно нулю на всей области параметров.

Заключение. В предложенной статье рассмотрена семантика плоских многопотоковых моделей на языке *UCM*. Представленное описание позволяет использовать дедуктивные средства верификации моделей на языке *UCM*, что существенно уменьшает количество генерируемых трасс и дает возможность верификации модели для любых возможных сценариев.

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

1. *International* Telecommunications Union. Recommendation Z.151 – User Requirements Notation (URN). – 2008. – 208 p.
2. *International* Telecommunications Union. Recommendation Z.120 – Message Sequence Charts (MSC). – 1998. – 136 p.
3. *Object* Management Group. Unified Modeling Language Specification, 2.0 – 2003. – 206 p.
4. *International* Telecommunications Union. Recommendation Z.100 – Specification and Description Language (SDL). – 1999. – 246 p.
5. *Kealey J.*, *Amyot D.* Enhanced Use Case Map Traversal Semantics // 13th SDL Forum (SDL'07), Paris, France, Springer, LNCS 4745. – Sept. 2007. – P. 133–149.
6. *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 Computer Science. – Ottawa, Canada, University of Ottawa, Sept. 2010.

Таблица 5. Семантика конструктов *WaitingPlace* и *Timer*

№	Описание	Изображение	Уравнение	Комментарий
25	Ожидание без пути триггера		S = check(c).T	S – WaitingPlace, c – условие
26	Транзитивное ожидание с путем триггера из пустого конструкта		S = check(c & (TransientTrigger(S) = true)).apply(TransientTrigger(S) := false).T	S – WaitingPlace, c – условие
27	Персистентное ожидание с путем триггера из пустого конструкта		S = check(c & (PersistentTrigger(S) > 0)).apply(PersistentTrigger(S) := PersistentTrigger(S) – 1).T	S – WaitingPlace, c – условие
28	Транзитивное ожидание с путем триггера из конца пути		S = check(c & (TransientTrigger(S) = true)).apply(TransientTrigger(S) := false).T	S – WaitingPlace, c – условие
29	Персистентное ожидание с путем триггера от конца пути		S = check(c & (PersistentTrigger(S) > 0)).apply(PersistentTrigger(S) := PersistentTrigger(S) – 1).T	S – WaitingPlace, c – условие
30	Переход к таймеру с путем триггера из пустого конструкта и без пути исчерпания		S = T	T – Timer
31	Переход к таймеру с путем триггера от конца пути и без пути исчерпания		S = T	T – Timer
32	Переход к таймеру без пути триггера и с путем исчерпания		S = apply(timer(T) := timer(T) + 1).T	T – Timer
33	Переход к таймеру с путем триггера из пустого конструкта и с путем исчерпания		S = apply(timer(T) := timer(T) + 1).T	T – Timer
34	Переход к таймеру с путем триггера с конца пути и с путем исчерпания		S = apply(timer(T) := timer(T) + 1).T	T – Timer
35	Переход от таймера без путей триггера и исчерпания		S = check(c).T	S – Timer, c – условие
36	Переход от транзитивного таймера с путем триггера из пустого конструкта и без пути исчерпания		S = check(c & (TransientTrigger(S) = true)).apply(TransientTrigger(S) := false).T	S – Timer, c – условие
37	Переход от персистентного таймера с путем триггера из пустого конструкта и без пути исчерпания		S = check(c & (PersistentTrigger(S) > 0)).apply(PersistentTrigger(S) := PersistentTrigger(S) – 1).T	S – Timer, c – условие
38	Переход от транзитивного таймера с путем триггера от конца пути и без пути исчерпания		S = check(c & (TransientTrigger(S) = true)).apply(TransientTrigger(S) := false).T	S – Timer, c – условие
39	Переход от персистентного таймера с путем триггера с конца пути и без пути исчерпания		S = check(c & (PersistentTrigger(S) > 0)).apply (PersistentTrigger(S) := PersistentTrigger(S) – 1).T	S – Timer, c – условие
40	Переход от таймера без пути триггера и с путем исчерпания		S = check(c2).apply(timer(S) := timer(S) – 1).T2 + check(~c2 & c1 & (timer(S) > 0)).apply(timer(S) := timer(S) – 1).insert(T1).S	S – Timer, c1, c2 – условия
41	Переход от транзитивного таймера с путем триггера из пустого конструкта и с путем исчерпания		S = check(c2 & (TransientTrigger(S) = true)).apply (timer(S) := timer(S) – 1, TransientTrigger(S) := false).T2 + check(~(c2 & TransientTrigger(S) = true)) & c1 & (timer(S) > 0).apply(timer(S) := timer(S) – 1).insert(T1).S	S – Timer, c1, c2 – условия
42	Переход от персистентного таймера с путем триггера из пустого конструкта и с путем исчерпания		S = check(c2 & (PersistentTrigger(S) > 0)).apply (timer(S) := timer(S) – 1, PersistentTrigger(S) := PersistentTrigger(S) – 1).T2 + check(~(c2 & (PersistentTrigger(S) > 0)) & c1 & (timer(S) > 0)).apply(timer(S) := timer(S) – 1).insert(T1).S	S – Timer, c1, c2 – условия
43	Переход от транзитивного таймера с путем триггера от конца пути и с путем исчерпания		S = check(c2 & (TransientTrigger(S) = true)).apply(timer(S) := timer(S) – 1, TransientTrigger(S) := false).T2 + check(~(c2 & TransientTrigger(S) = true)) & c1 & (timer(S) > 0)).apply(timer(S) := timer(S) – 1).insert(T1).S	S – Timer, c1, c2 – условия
44	Переход от персистентного таймера с путем триггера от конца пути и с путем исчерпания		S = check(c2 & (PersistentTrigger(S) > 0)).apply (timer(S) := timer(S) – 1, PersistentTrigger(S) := PersistentTrigger(S) – 1).T2 + check(~(c2 & (PersistentTrigger(S) > 0)) & c1 & (timer(S) > 0)).apply(timer(S) := timer(S) – 1).insert(T1).S	S – Timer, c1, c2 – условия

Окончание на стр. 34.

7. *Guan R*. From Requirements to Scenarios through Specifications: A Translation Procedure from Use Case Maps to LOTOS. M.Sc. thesis, OCICS, University of Ottawa, Canada, 2002.
8. *Hassine J*., *Rilling J*., *Dssouli R*. An ASM Operational Semantics for Use Case Maps // 13 th IEEE Int. Requirement Engin. Conf. (RE'05), IEEE Comp. Society Press, Sept. 2005. – P. 467–468.
9. *Jameleddine Hassine*. Formal semantics and verification of use case maps // A thesis in The Department of Comp. Sci. and Software Eng., Montreal, Quebec, Canada, Concordia University, 2008. – 299 p.
10. *Weiss M*., *Amyot D*. Business Process Modeling with URN // Int. J. of E-Business Research. – July-Sept. 2005. – **1**(3). – P. 63–90.
11. *Workflow* Control-Flow Patterns: A Revised View. BPM Center Report / Nick Russell, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst et al. / BPM Center Report BPM-06-22. – BPMcenter.org, 2006. – 134 p.
12. *Wil M.P. van der Aalst ter Hofstede*, *Kiepuszewski B*., *Barros A.P*. Workflow Patterns. Distributed and Parallel Databases. – July 2003. – **14**(3). – P. 5–51.
13. *Peter Huber*, *Kurt Jensen*, *Robert M. Shapiro*. Hierarchies in coloured petri nets // Proc. of the 10th Int. Conf. on Applications and Theory of Petri Nets. – London: Springer-Verlag, UK, 1991. – P. 313–341.
14. *ISO*/*IEC*. Software and system engineering. High-level petri nets, part 1, Concepts, definitions and graphical notation, 2004. – 38 p.
15. *Letichevsky A*., *Gilbert D*. A Model for Interaction of Agents and Environments / D. Bert, C. Choppy, P. Moses (Eds) // Resent Trends in Algebraic Development Techniques, LNCS 1827, 1999. – P. 311–328.
16. *Годлевский А.Б.* Инсерционная семантика параллельных процедурных конструктов языка *UCM* // УСиМ. – 2012. – № 6. – С. 22–34.

Тел. для справок: +38 044 526-0058, +38 044 200-8424 (Киев)
E-mail: *antonguba@ukr.net*, *costa@iss.org.ua*