УДК 681.3

С.С. ГОРОХОВСКИЙ, А.Н. ГЛИБОВЕЦ, О.А. ПОПОВ

К ФОРМАЛИЗАЦИИ АГЕНТНО-ОРИЕНТИРОВАННЫХ СИСТЕМ

Ключевые слова: *агенты, агентно-ориентированные системы, формальные модели агентов.*

ВВЕДЕНИЕ

В течение нескольких лет технология мобильных агентов остается популярной темой в сфере информационной инженерии [1]. На эту технологию возлагаются большие надежды, но широкомасштабное применение агентов видится пока еще в будущем. Одной из причин несовершенства и незрелости технологии агентов является то, что мы все еще неспособны создать эту технологию достаточно эффективной и такой, которая соответствовала бы жестким требованиям практического применения.

Мобильные агенты (МА) — это программы с постоянными свойствами и признаками, которые самостоятельно передвигаются в сети и могут общаться с окружением или другими агентами. Они используют специализированные серверы для интерпретации поведения агентов и коммуникации с другими серверами. МА могут выполняться на какой-либо машине в сети без наличия предварительно инсталлированного кода агента на машине, на которую может зайти агент. МА представляют новую модель в процессе эволюции исполнительного наполнения в Интернете, используя программный код, который может перемещаться вместе со статической информацией (информацией о состоянии канала) [2].

МА используются во многих прикладных программах и вычислительных средах. Мотивация такого распространения имеет несколько причин: МА осуществляют эффективный асинхронный метод поиска информации или услуг в стремительно эволюционной сети; они могут быть запущены в неструктурированную сеть и, перемещаясь в ней, собирать информацию. МА могут функционировать и при ненадежном соединении в медленных сетях с помощью простых устройств. Поэтому они имеют много преимуществ в системном программировании для Интернета [3], в котором существует реальная потребность оптимизации интеграции информации, мониторинга, безопасности и надежности [4]. Парадигма МА успешно проявила себя и в реализации распределенного доступа к сетевым базам данных за счет прозрачного механизма распространения найденной и отфильтрованной информации, а также возможности минимизации загрузки сети [5, 6].

Кроме того, МА эффективно и асинхронно выполняют команды клиента при ненадежном соединении и динамичной адаптации к различным типам соединения, что является обычным в беспроводных средах [7].

Имеется много разных агентно-ориентированных (АО) методологий. В основу их разработки положены идеи реализации онтологий или используемые в объектно-ориентированных (ОО) методологиях, или идеи их специфического объединения [8].

1. ФОРМАЛЬНЫЕ МОДЕЛИ АГЕНТОВ

Подобно разработке традиционного программного обеспечения для проектирования и реализации эффективных агентных систем важно иметь определенный

формальный способ описания и спецификации поведения таких систем, т.е. нужны формальные модели агентов [1, 9]. Сложность построения моделей обусловлена и отсутствием точного формального языка представления агентов и агентной архитектуры. Даже когда агентные системы представлены с помощью формальной семантики, разнообразие используемых стилей и формализмов никак не способствуют их сопоставлению [10–12].

Формальные теории агентов можно рассматривать, как спецификацию агентов, которая не только описывает, определяет и ограничивает поведение агента, но и является так называемой «спецификацией программной системы» в современной программной инженерии. Они обеспечивают базис проектирования, реализации и верификации агентных систем. Агенты, естественно, являются следующей ступенью программной инженерии. Они представляют фундаментально новый взгляд на сложные распределенные системы.

2. КОНЦЕПЦИЯ АГЕНТОВ КАК РАЗВИТИЕ КОНЦЕПЦИИ ОБЪЕКТА

Для моделирования многих сложных информационных систем применяют ОО технологии. Однако концепция «объекта» (по крайней мере, в его обычном смысле) не покрывает все аспекты современных информационных систем. Используя ОО подход, легко описать структуру таких систем, но динамическое поведение такой системы можно описать только незначительно. Поэтому нужна семантическая модель, где спецификация поведения объекта или объектной системы может меняться на протяжении их существования. До сих пор это невозможно было выразить с помощью существующих ОО языков спецификаций. Концепцию агента можно рассматривать как дальнейшее развитие концепции объекта для моделирования динамики информационных систем. В отличие от традиционных объектов агенты могут менять свое поведение динамично во время непрерывного функционирования систем, т.е. поведение агента не определено и не может быть определено на этапе спецификации или компиляции.

Агента можно определить как автономный активный объект. Концепция автономии значительно шире, чем определение объекта, обычные вызовы методов не типичны для агентов [13]. Для спецификации динамического поведения агента можно использовать расширение одного из имеющихся ОО языков спецификаций, а для формализации такого описания — расширенную темпоральную логику.

2.1. Переход от спецификации объекта к его проектированию. Объект можно охарактеризовать как связную структурную единицу поведения. Он имеет внутреннее состояние, определяемое атрибутами, которое можно изменять через интерфейс событий. Объекты можно рассматривать и как наблюдаемый процесс. Атрибуты суть наблюдаемые свойства объектов, которые можно изменять только во время выполнения событий.

Поведение объектов можно описать жизненными циклами (или трассами), которые построены из последовательности (или множества одновременно выполняемых) событий. Соответственно каждое состояние объекта полностью характеризуется его снимком событий, который определяет текущие значения атрибутов. Возможная эволюция объекта может быть ограничена определением множества ограничений на состояния и будет использоваться для предоставления допустимых переходов состояния для объекта.

Для текстуального представления спецификации объекта можно использовать язык спецификаций Troll (рис. 1) [14].

```
1:object class Account
2:  identification        ByAccountID: (Bank, No);
3:  attributes            No: nat constant;
4:        Bank:           |Bank|;
5:        Holder:         |Customer|;
6:        Balance:        money initialized 0.00;
7:        Limit:          money initialized 0.00;
8:        Counter:        nat initialized 0;
9:  events
10:    Open (BID:|Bank|, AccNo:nat, AccHolder:|Customer|) birth
11:              changing Bank   := BID,
12:                       No     := AccNo,
13:                       Holder := AccHolder;
14:     Withdraw(W:money) enabled Balance - Limit > = W
15:                       changing Balance := Balance - W
16:                       calling IncreaseCounter;
17:     Deposit(D:Money)  changing Balance := Balance + D
18:                       calling IncreaseCounter;
19:      IncreaseCounter  changing Counter := Couner + 1
20:  Close death;
21:end object class Account;
```

Рис. 1. Спецификация класса *Account* на языке Troll

В Troll-подобных языках шаблон спецификации объекта состоит из двух разделов: сигнатуры, в которой перечислены события и атрибуты объекта, и поведенческой составляющей, содержащей аксиомы [6].

Предложенная спецификация объекта имеет один серьезный недостаток: она позволяет лишь частично определить динамичное поведение информационной системы. Все возможные модификации объекта должны быть зафиксированы на этапе спецификации, т.е. перед созданием объекта. Это является очень существенным ограничением для поддержки эволюции объекта в унифицированной системе.

Обычно при существовании объекта в системе может возникнуть потребность изменить объект или среду в такой способ, который невозможно было предусмотреть заблаговременно. Возникает потребность в среде, где описание объекта может быть изменено во время работы системы. Поэтому агента желательно рассматривать как интеллектуальный объект, постоянно эволюционирующий, который на базе знания может делать выводы, динамически изменять состояние и поведение. Агенты, как и объекты, имеют внутреннее состояние, которое отражает их историю и определяет поведение. Поскольку внутреннее состояние объектов определяется значением его атрибутов, агенты имеют более общее определение состояний. Кроме значений атрибутов, они могут содержать альтернативную информацию, частичные знания, предположения и др.

Собственно внутреннее состояние агента отражает знания (верования, намерения, обязательства и др.) в определенный момент времени. В отличие от традиционной объектной концепции эти знания не фиксированы на этапе спецификации объекта — они могут изменяться во время жизненного цикла агента. Можно утверждать, что внутреннее состояние объекта содержит строгие знания (зафиксированные в момент создания объекта и не пересматриваемые), а также и некоторые переменные знания (которые можно рассмотреть или изменить с учетом ограничений во время эволюции агента).

2.2. Агентно-ориентированный язык спецификаций. Язык спецификаций АО систем должен давать возможность определять динамическое поведение. Начнем с того, что будем считать состояния теориями. Это даст возможность, в отличие от привычного ОО подхода, где состояние объекта можно описать простой картой значений (в которой для каждого атрибута определено определенное значение), применить набор формул для описания состояния.

Заменяем обычные изменения состояния на изменения теории, с помощью которых можно описывать знания или задания агента. Таким образом, можно специфицировать пересмотр существующих знаний и приобретение новых, а также определить частичные знания и знания по умолчанию.

Предлагается двухуровневая система для моделирования информационных систем в терминах агентов. На первом уровне содержатся обычные атрибуты и события, описывающие фиксированное поведение агента. На втором уровне специфицирована возможная эволюция агента.

На рис. 2 изображена возможная спецификация агента *Account*. Часть спецификации *rigid axioms* никогда не должна меняться. В нашем примере — это результат выполнения таких операций, как *Withdraw* и *Deposit* над атрибутом *Balance.*

К спецификации объекта добавим еще одну концепцию: агент будет иметь аксиоматические атрибуты, правильные при некоторых условиях. В нашем примере аксиоматический атрибут *Axioms* инициализирован пустым множеством аксиом. При наличии нескольких таких атрибутов необходимо явно маркировать один из них как текущий набор аксиом. Каждая формула, которая включается в определенный аксиоматический атрибут в определенном состоянии, должна выполняться в этом состоянии.

Как и простые атрибуты, аксиоматические атрибуты можно изменять с помощью мутаторов. Такое изменение — специальный вид событий. В данном примере мы можем манипулировать аксиоматическим атрибутом *Axioms*. Можно прибавлять новые аксиомы к *Axioms*, убирать существующие или возвращать *Axioms* к первоначальному состоянию.

```
 1:agent class Account
 2:  identification        ByAccountID: (Bank, No);
 3:  attributes    No: nat constant;
 4:     Bank:      |Bank|;
 5:     Holder:    |Customer|;
 6:     Balance:   money initialized 0.00;
 7:     Limit:     money initialized 0.00restricted >= -5000.00;
 8:     Counter:   nat initialized 0;
 9:  events
10:  Open (BID:|Bank|, AccNo:nat, AccHolder:|Customer|) birth;
11:          Withdraw(W:money);
12:          Deposit(D:Money);
13:          IncreaseCounter;
14:          Close death;
15:  rigid axioms
16:  Open (BID:|Bank|, AccNo:nat, AccHolder:|Customer|) birth
17:                changing Bank   := BID,
18:                         No     := AccNo,
19:                         Holder := AccHolder;
20:  Withdraw(W:money) enabled Balance - Limit > = W
21:                    changing Balance := Balance - W
22:                    calling IncreaseCounter;
23:  Deposit(D:Money)  changing Balance := Balance + D
24:                    calling IncreaseCounter;
25:  IncreaseCounter   changing Counter := Couner + 1
26:  axiom attributes       Axioms intialized {};
27:  mutators               ResetAxioms;
28:                         AddAxioms(P:Formula);
29:                         RemoveAxioms(P:Formula);
30:  dynamic specification ResetAxioms  changing Axioms := {};
31:               AddAxioms(P) changing Axioms := Axioms U P;
32:               RemoveAxioms(P) changing Axioms := Axioms - P;
33: end agent class Account;
```

Рис. 2. Спецификация агента *Account* на расширенном языке Troll

2.3. Формализация. Для формализации представленной выше спецификации, используем темпоральную логику. Пусть $o.Attr = v$ означает, что атрибут *Attr* объекта *o* имеет значения v, а $o.\nabla e$ означает, что у объекта *o* происходит событие *e*.

Для создания выражений используем обычные операции: $\neg$ (отрицание), $\wedge$ (конъюнкцию) и др., которые можно выразить этими операторами. При этом допустим использование темпоральных операторов: $\circ$ (последующий), $\Box$ (всегда в будущем) и $\Diamond$ (иногда в будущем $\Diamond f \equiv \neg \Box \neg f$).

Опишем события *Open* и *Transfer*, определенные в предыдущем разделе, для объекта агента *Account*:

$$\Box (a.\nabla Open(B, N, H) \to \circ (a.Bank = B \wedge a.No = N \wedge a.Holder = H)).$$

Поскольку *Open* — конструктор объекта, он вызывается только один раз за время «жизни» объекта:

$$\Box (a.\nabla Open(B, N, H) \to \circ \Box \neg (\exists B', N', H' : a.\nabla Open(B', N', H'))).$$

Событие *Transfer* можно выразить так:

$$\Box b.\nabla Transfer(A_1, A_2, M) \to$$
$$\to (Account(A_1).\nabla Withdraw(M) \wedge Account(A_2).\nabla Deposit(M)).$$

Заметим, что довольно сложно выразить изменения состояний агента в терминах линейной темпоральной логики.

3. ЯЗЫК Gamma

Для полноты спецификации агентов нерешенной остается задача спецификации взаимодействия между агентами. На наш взгляд, перспективным в этом плане является использование языка Gamma [15].

Gamma — это язык программирования высокого уровня, в котором используются преобразования мультимножеств, а параллелизм определяется неявно. В соответствии с парадигмой Gamma программисты должны сосредоточиться на логике передачи кортежей (это терминология заимствована из Linda) между взаимодействующими процессами. Семантика Gamma базируется на метафоре химической реакции.

3.1. Основные концепции языка. В распределенной системе сущности распределены в пространстве и взаимодействуют между собой через коммуникационные связи. Такие системы создавать сложно. Проектировщик системы старается уменьшить накладные расходы, связанные с декомпозицией всей системы, на большое множество элементарных задач. Поэтому для проектирования таких систем предложено использовать язык Gamma.

Программа, написанная на Gamma, состоит из наборов правил, которые регламентируют взаимодействие между программными модулями. Эти наборы правил называются «реакциями» — по аналогии с химическими реакциями. Основные структуры данных моделируются с помощью мультимножеств. При этом отсутствует глобальный контроль за порядком элементов мультимножества, которые участвуют в реакции. Выполнение программы продолжается до тех пор, пока реакция не прекратится. Понятно, что результатом вычислений также есть определенное мультимножество.

Мультимножество — это обыкновенное множество, однако его элементы здесь могут повторяться. Последовательность вычислений в программе есть переход от первичного множества к заключительному. Элементы мультимножества называются кортежами.

Мультимножества — это единственная структура данных, а программы описываются как совокупность пар (условие реакции, действие). Например, следующая Gamma-программа вычисляет максимальный элемент непустого множества:

$$\max\ M = x : M,\ \ y : M \to x : M \to x \geq y.$$

Условие $x \geq y$ реакции определяет свойство, которому должны удовлетворять избранные элементы x и y. При этом порядок вычислений в момент сравнения элементов не фиксируется. Если несколько несвязанных пар элементов отвечают условиям, то сравнения и изменения могут происходить параллельно. Интуитивно способ работы на языке Gamma можно описать с помощью метафоры химической реакции: вычисления заканчиваются по достижении стабильного состояния или при отсутствии элементов, удовлетворяющих условию реакции. Иногда программы могут не заканчиваться.

В концепции «Gamma высшего уровня» программа определяется как конфигурация $[P, E]$, где P — программа, а E — набор именованных мультимножеств, который можно рассматривать как среду программы. Каждый компонент среды — типизованное мультимножество. Программа изымает элементы из этих мультимножеств и производит новые элементы. Например, программа

$$x : M,\ pivot : Pivot \to x : R_{\leq},\ pivot : Pivot \leftarrow x \leq pivot$$

изымает элемент x из мультимножества M и $pivot$ из мультимножества $Pivot$, и как только выполняется условие $x \leq pivot$, x изымается из M и прибавляется к $R_{\leq}$, а $pivot$ остается в $Pivot$.

Программы можно создавать, используя последовательный или параллельный оператор «+». Соответственно семантике операции Gamma среда, образованная параллельной композицией $P_1 + P_2$, должна быть стабильной для P_1 и P_2 одновременно. Когда же рассматривают последовательную композицию P_1 и P_2, то начальная среда для P_1 должна быть стабильной средой для P_2, а окончательная среда возвращается с P_1.

Например, программа, сравнивающая элементы мультимножества целых чисел M с $pivot$ и распределяющая элементы по $R_{\leq}$ и $R_{>}$, имеет вид

$Part\ M_0 = [P,\ M = M_0,\ R_{\leq} = \Phi,\ R_{<} = \Phi,\ Pivot = \{p\}]$ where
$P = P_1 + P_2$
$P_1 = x : M,\ pivot : Pivot \to x : R_{\leq},\ pivot : Pivot \leftarrow x \leq pivot$
$P_2 = x : M,\ pivot : Pivot \to x : R_{>},\ pivot : Pivot \leftarrow x > pivot$

3.2. Моделирование агентных коммуникаций и миграции агентов. Посредством языка Gamma можно лаконично описать модель вычислений. Однако для моделирования агентных систем необходимо найти механизм описания агентной коммуникации и миграции агентов. Если моделировать агенты на языке Gamma, то агентную коммуникацию и миграцию агентов можно рассматривать как обмен сообщениями между агентами. Следующее правило описывает передачу сообщения M или миграцию агента M (когда M — программа):

$$M : E_1,\ \ M' : E_2 \to [M, M'] : E_2 \leftarrow L(E_1, E_2) \wedge C(M, M_2),$$

где E_1 и E_2 — две разных среды, $L(E_1 > E_2)$ — коммуникационная связь между E_1 и E_2, а $C(M, M')$ — условие, при котором происходит передача сообщения или миграция агента.

Примеры описания отдельных типов агентных систем на языке Gamma можно найти в [16]. Для примера рассмотрим понятие коллаборативного агента. Коллаборативные агенты решают задачи, сотрудничая между собой. Информация уточняется в процессе этого сотрудничества [17].

Рассмотрим систему оценки студенческих проектов в on-line. Система состоит из таких модулей: сбор проектов, оценка проектов, студенческих записей, и базы данных проектов, сохраняющей проекты, присланные во время прохождения курса. Процесс оценки проектов регулируют такие правила: модуль сбора проектов собирает студенческие работы; если присланная работа есть плагиат (определяется по работам из базы данных проектов), то в студенческую запись этого студента добавляется оценка «*F*»; иначе проект передается модулю оценок и оценивается, оценка добавляется к записям студента, а проект добавляется в базу данных проектов; все модули работают параллельно.

Пусть система имеет четыре модуля: модуль сбора проектов (*Col*), модуль оценки проектов (*Eva*), модуль управления базой данных проектов (*Dbm*), модуль поддержки студенческих записей (*Srm*). В процессе использованы такие типы данных: *Prg* — студенческий проект перед его подачей на рассмотрение в форме $(id,\ p)$, где *id* — идентификатор студента, p — проект; *Sub* — присланные проекты в форме $(id, n,\ p)$, где n — натуральное число, которое помечает следующий проект для сравнения; *Db* — проекты, сохраненные в базе данных; *Rec* — студенческие записи в форме (id, g), где *id* — идентификатор студента, g — оценка.

Для моделирования коммуникаций между агентами использованы такие типы данных: *Qcd* — сообщение в очереди из *Col* в *Db*, сообщение посылается из *Col* в *Db*, когда получаем новое сообщение, тогда полученный проект нужно сравнить с предварительно сохраненными проектами; *Qdc* — сообщения в очереди из *Db* в *Col*, которое содержит сохраненные для следующего сравнения проекты; *Qcs* — сообщения в очереди из *Col* в *Srm*, если выявлен плагиат и в студенческие записи посылается оценка; *Qce* — сообщение в очереди из *Col* в *Eva*, после выполнения сравнения проект посылается в *Eva* для оценивания; *Qes* — сообщение в очереди из *Eva* в *Srm* после оценивания, в *Srm* посылается оценка; *Qed* — сообщение в очередь из *Eva* в *Db*, после выполнения оценки проект посылается в *Db* для регистрации и дальнейшего сохранения.

Такую систему на языке Gamma можно описать так:

$$
\begin{aligned}
&ProjEva\ \ M = [P, Prj = M, Sub = \Phi, Db = D_0, Rec = \Phi,\\
&\qquad Q_{cd} = \Phi,\ Q_{dc} = \Phi, Q_{cs} = \Phi,\ Q_{ce} = \Phi,\ Q_{es} = \Phi,\ Q_{ed} = \Phi]\ ,\ where\\
&\quad P = Col + Eva + Dbm + Srm\\
&\qquad Col = Q_1 + Q_2 + Q_3 + Q_4\\
&\qquad\quad Q_1 = (id,\ p) : Prj \to (id\ , l,\ p) : Sub,\ \ (id, 0) : Q_{cd} \leftarrow Ready(p)\\
&\qquad\quad Q_2 = (id, n,\ p) : Sub,\ \ (n, s) : Q_{do} \to (id, F) : Q_{cs} \leftarrow p = s\\
&\qquad\quad Q_3 = (id, n,\ p) : Sub,\ \ (n, s) : Q_{do} \to (id, n+1,\ p) : Sub \leftarrow p\,! = s\\
&\qquad\quad Q_4 = (id, n,\ p) : Sub, (n, s) : Q_{do} \to (id,\ p) : Q_{ce}\\
&\qquad Dbm = Q_5 + Q_6 + Q_7\\
&\qquad\quad Q_5 = (id, n) : Q_{cd}\,(n+1, s) : Db \to (id, n+1) : Q_{cd}\,(n+1, s) : Q_{do}\,(n+1, s):\\
&\qquad\quad Db \leftarrow x < Size(Db)\\
&\qquad\quad Q_6 = (id, n) : Q_{cd} \to (n, id) : Q_{dc} \leftarrow n \ge Size(Db)\\
&\qquad\quad Q_7 = p : Q_{ed}\,(n, s) : Db \to (n, s) : Db, (n+1,\ p) : Db \leftarrow n = Size(Db)\\
&\qquad Eva = Q_8\\
&\qquad\quad Q_8 = (id,\ p) : Q_{cs} \to (id, g) : Q_{es}\,p : Q_{ed} \leftarrow g = Evaluate(p)\\
&\qquad Srm = Q_9\\
&\qquad\quad Q_9 = (id, g) : Q_{es} \to (id, g) : Rec
\end{aligned}
$$

4. КООРДИНАЦИОННАЯ МОДЕЛЬ

Для системы, состоящей из одного агента, главным является поведение и интерфейс этого агента. Однако для мультиагентной системы особенно важна координационная модель, учитывающая все аспекты взаимодействия агентов. Чтобы проектирование мультиагентной системы было эффективным, нужно сосредоточиться на роли и свойствах механизма координации (абстракции коммуникации) внутри координационной модели, а не на объектах координации (агентах).

Для эффективного проектирования мультиагентной системы используем коммуникационный механизм, подобный *Linda*. Модель *ACLT* [18] — это расширение *Linda*. Пространство картежей *ACLT*, в отличие от пространства картежей *Linda*, реагирует не только на перемену состояния, но и на коммуникационные события, т.е. кортежи *ACLT* являются программированными. Реакцию на коммуникационные события определим с помощью специализированного языка спецификаций. Программируемость пространства картежей *ACLT* дает возможность реализовать разные координационные политики агентов, не модифицируя при этом поведение конкретного агента.

4.1. Абстрагирование коммуникаций. В модели *ACLT* коммуникация происходит с помощью разнообразных именованных пространств кортежей как наборов логических высказываний первого порядка, каждое из которых уникально идентифицирует базовый терм. Логическое пространство кортежей может иметь двойную интерпретацию: простой коммуникационный механизм или хранилище знаний. Поэтому о коммуникационных состояниях можно сделать дедуктивный вывод.

4.2. Модель реакции. Основной идеей модели *ACLT* есть идея определения множества логических событий (каждое из которых имеет уникальное имя), ассоциированных с физическими (коммуникационными) событиями. Несколько физических событий могут соответствовать одному логическому событию, и несколько логических событий можно присоединить к одному логическому событию. Ассоциация между коммуникационным и логическим событиями задается кортежем *map*(*Operation*, *Event*). Иначе говоря, каждый раз, когда выполняется операция (*Operation*) над пространством картежей, происходит логическое событие (*Event*).

Логические события можно ассоциировать с реакциями, инициированными определенным событием. Поведение реакции определено через кортеж *react*(*Event*, *Goal*), где цель реакции (*Goal*) — это набор примитивных операций, выполняемых в ответ на логическое событие (*Event*). Цель реакции — формула одного из приведенных ниже типов:

- коммуникационные примитивы без блокирования (out, in_noblock, rd_noblock, ...);
- примитивы состояния (current agent/1, current_tuple/1, current_op/1,...);
- терминальные предикаты (терм равенства/неравенства и др.)

4.3. Задача обедающих философов. Основное преимущество координационной модели *ACLT* — возможность менять координационную модель мультиагентной системы без внесения изменений в каждый конкретный агент. Для примера рассмотрим известную задачу о философах, которые обедают.

Пример

1. Необходимы две вилки одновременно: ***required*(F1, F2)**

```
1: map(in, hungry).
2: react(hugry, (current_tuple(forks(F1,F2)), pre,
3:                          out(required(F1,F2))
4:                          )).
```

```
5:react (hungry, (current_tuple(forks(F1,F2)), post
6:                        in_noblock(required(F1,F2))
7:                        )).
```

2. Каждая доступная вилка задается в форме кортежа ***fork*(*Fork*)**

```
8:map(out, thoughtful).
9:react( thoughtful, (current_tuple(release(F1,F2) )
10:                   out(fork(F1)),
11:                   out(fork(F2)),
12:                   in_noblock(release(F1,F2))
13:                   )).
```

3. Пространство кортежей запрограммировано так, чтобы удовлетворить запрос на вилки, как только вилки освободятся, или выполняется новый запрос.

```
14:map(out, reserve).
15:react(reserve, (current_tuple(required(F1,F2)),
16:                        in_noblock(fork(F1))
17:                        in_noblock(fork(F2))
18:                        out(forks(F1,F2))
19:)).
20:react(reserve, (current_tuple(fork(F)),
21:                        rd_noblock(required(F1,F))
22:                        in_noblock(fork(F1))
23:                        in_noblock(fork(F))
24:                        out(forks(F1,F))
25:)).
26:react(reserve, (current_tuple(fork(F)),
27:                        rd_noblock(required(F,F2))
28:                        in_noblock(fork(F))
29:                        in_noblock(fork(F2))
30:                        out(forks(F,F2))
31:)).
```

Из примера видно, что существенно изменить коммуникационную политику в мультиагентной системе, описанной с помощью этой модели, можно без внесения изменений в программный код агентов.

5. СРАВНЕНИЕ МОДЕЛЕЙ Gamma И ACLT

Модели Gamma и *ACLT* используют понятие «реакция». Метафора химической реакции, используемая Gamma, позволяет описать общие законы координации в терминах условий реакции и последующих событий. Однако в модели Gamma отсутствует выделенное абстрактное коммуникационное пространство. Модель Gamma не учитывает наблюдаемые свойства агента, ее реакции — единственный способ эволюции мультиагентной системы. Реакции Gamma можно рассматривать как спецификацию, на высоком уровне регламентирующую пути эволюции мультиагентной системы и не зависящую от модели вычислений.

Реакции моделей *ACLT* отвечают точному, реальному поведению системы.

ЗАКЛЮЧЕНИЕ

В работе проанализировано использование формальних моделей для проектирования агентных систем. Значительное внимание уделено сравнению моделей Gamma и *ACLT*. Практическая значимость работы состоит в создании основ

инструментария для динамичного изменения поведения объекта, контроля его логической и функциональной целостности средствами аспектно-ориентированного и объектно-ориентированного программирования.

СПИСОК ЛИТЕРАТУРЫ

1. Анісімов А.В., Глибовець А.М., Жаб'юк В.Я. Основні архітектурні принципи побудови програмних систем реалізації мобільних агентів // Вісн. Київ. нац. ун-ту імені Тараса Шевченка. Сер.: фіз.-мат. науки. — 2008. — Вип. 3. — С. 125–131.
2. Гороховський С.С. Агентні технології: спроба критичного огляду // Наукові записки НУ «Києво-Могилянська Академія». — Т. 18: Спец. випуск. — К.: НаУКМА, 2000. — С. 391–395.
3. Lange D., Oshima M. Seven good reasons for mobile agents // Commun. of the ACM. — 1999. — **42.** — P. 88–89.
4. Samaras G., Evripidou P., Pitoura E. Mobile-agents based Infrastructure for eWork and eBussiness applications // Proc. of the eBusiness and eWork Conf. — 2000. — P. 1092–1098.
5. Papastavrou S., Samaras G., Pitoura E. Mobile agents for WWW distributed database access // IEEE Transact. on Knowledge and Data Engineer. Journ. (TKDE). — 2000. — **12**, N 5. — P. 802–820.
6. Villate Y., Illarramendi A., Pitoura E. Data lockers: mobile-agent based middleware for the security and availability of roaming users data // Proc. of 5th Intern. Conf. on Cooperative Inform. Systems (CoopIS'2000). — 2000. — P. 275–286.
7. Samaras G., Pitsillides A. Client/Intercept: a computational model for wireless environments // Proc. of the 4th Intern. Conf. on Telecomm. (ICT'97). — 1997. — P. 1205–1210.
8. Henderson-Sellers B., Giorgini P. Agent-oriented methodologies. — PA.: Idea Group Publ., 2005. — 428 p.
9. Meyer J., Schobbens P. Formal models of agents: an introduction // Lecture Notes in Comput. Sci. — 1999. — **1760.** — P. 1–7.
10. Инсерционное программирование / А.А. Летичевский, Ю.В. Капитонова, В.А. Волков, В.В. Вышемирский, А.А. Летичевский (мл.) // Кибернетика и системный анализ. — 2003. — № 1. — С. 12–32.
11. Letichevsky A.A., Gilbert D.R. A model for interaction of agents and environmenrs // Recent Trends in Algebra's Development Technique Language, 2000. — P. 311–329.
12. Hindriks K., d'Inverno M., Luck M. Architecture for agent programming languages // ECAI 2000: Proceedings of the Fourteenth European Conference on Artificial Intelligence. — http://autonomousagents.org/2001/oas.
13. Depke R., Heckel R., Kuster J. Improving the agentoriented modeling process with roles // Proc. 5th Intern. Conf. on Autonomous Agents (AGENTS-2001). Proc. of the ACM. — Canada, 2001. — 645 p.
14. Conrad S., Saake G., Türker C. towards an agent-oriented framework for specification of information systems // Lecture Notes in Comput. Sci. — 1999. — **1760**. — P. 57–60.
15. Lemetayer D. Higher-order multiset processing // DIMACS Series in Discrete Mathematics and Theoretical Comput. Sci. — 1994. — **18.** — P. 179–200.
16. Ally M., Lin F. Designing distributed environments with intelligent software agents. — PA, Hershey: Idea Group Publ., 2004. — P. 242–249.
17. Frohlich P., Mora I., Nejdl W., Schroeder M. Diagnostic agents for distributed systems // Lecture Notes in Comput. Sci. — 1999. — **1760.** — 173 p.
18. Denti E., Omicini A. Designing multi-agent systems around an extensible communication abstraction // Ibid. — 1999. — **1760.** — P. 80–82.

Поступила 06.04.2011