

**Предикатное описание метаграфовой модели данных**

**Predicate representation of metagraph data model**

- 1. Гапанюк Ю.Е. (Gapanuk Yu.E.), доцент кафедры «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана (Bauman Moscow State Technical University), gapyu@bmstu.ru**
- 2. Ревунков Г.И. (Revunkov G.I.), доцент кафедры «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана (Bauman Moscow State Technical University), revunkov@bmstu.ru**
- 3. Федоренко Ю.С. (Fedorenko Yu.S.), аспирант кафедры «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана (Bauman Moscow State Technical University), Fedyura1992@yandex.ru**

*Аннотация.* Статья посвящена предикатному описанию метаграфовой модели данных. Традиционно предикатное описание используется в логических языках программирования. Рассмотрена критика языков на основе предикатного описания и критика логической парадигмы программирования. Показано, что предикатная форма синтаксиса является достаточно удобной и используется в ряде современных языков программирования. Также показано, что логическую парадигму программирования нельзя признать устаревшей, так как она изоморфна функциональной парадигме, а многие недостатки классического Пролога исправлены в современных логических языках. Рассмотрена метаграфовая модель данных. В качестве активного элемента метаграфовой модели используются метаграфовые агенты. Показано, как метаграфовая модель данных, включая агентов, может быть преобразована в предикатное описание. Важным свойством предложенного описания является

самоотображаемость, что позволяет агентам модифицировать как метаграфовые данные, так и структуру агентов нижнего уровня.

*Abstract.* The article is devoted to the predicate representation of metagraph data model. Traditionally, predicate representation is used in logical programming languages. The criticism of programming languages based on predicate syntax and the criticism of logical programming paradigm are discussed. It is shown that the predicate syntax is convenient enough and it is used in several modern programming languages. It is also shown that logical programming paradigm cannot be considered obsolete, since it is isomorphic to the functional programming paradigm, and many of the shortcomings of the classical Prolog language are fixed in modern logical languages. The metagraph data model is represented. Metagraph agents are used as the active element of metagraph data model. It is shown that metagraph data model, including the agents, can be converted to a predicate representation. An important property of the proposed representation is homoiconicity, which means that agents can modify metagraph data model as well as the structure of the lower level agents.

*Расширенный реферат научной статьи на русском языке:*

Статья посвящена предикатному описанию метаграфовой модели данных. Традиционно предикатное описание используется в логических языках программирования. В настоящее время существует мнение, что подход на основе предикатного описания и логические языки на основе этого подхода являются устаревшими.

Тем не менее, классическая предикатная форма является достаточно удобной для описания, так как напоминает вызов функций в большинстве языков программирования. Синтаксис предикатной формы существенно менее разнообразен, чем синтаксис функциональных языков программирования, что упрощает изучение языков на основе предикатной формы. Однако отсутствие именованных параметров следует отнести к недостаткам классической предикатной формы.

Мнение о том, что логические языки программирования являются устаревшими, опровергается, прежде всего, принципом изоморфизма Карри-Ховарда. Этот принцип устанавливает соответствие между системами типов (которые применяются в функциональных языках) и логическими исчислениями (которые являются основой логических языков). Таким образом, можно говорить об изоморфизме логической и функциональной парадигм программирования. Поэтому мнение о превосходстве одной из парадигм является некорректным.

Механизм поиска с возвратом (backtracking), применяемый в Прологе, обеспечивает удобный способ формирования запросов к базе знаний. Классический Пролог реализует предположение о замкнутости мира, однако современные логические языки, такие как AnsProlog, реализуют предположение об открытости мира. Классический Пролог не поддерживает исчисление предикатов высших порядков, однако его поддерживают современные логические языки, такие как Flora-2. Классический Пролог не поддерживает прямой логический вывод, в качестве расширения предлагается использование прямого логического вывода на метаграфах с использованием метаграфовых агентов. Пролог используется в современных проектах, связанных с искусственным интеллектом, таких как GATE (Пролог используется для конструирования синтаксических деревьев и обработки предложений) и HypergraphDB (диалект TuProlog используется как средство обработки гиперграфовой модели данных).

В данной статье в качестве модели данных предлагается использовать метаграфовую модель данных. Основными элементами этой модели являются вершина, ребро, метавершина, метаребро. Все элементы метаграфа аннотированы атрибутами.

Наличие у метавершин собственных атрибутов и связей с другими вершинами является важной особенностью метаграфов. Это соответствует принципу эмерджентности, то есть приданию понятию нового качества, несводимости понятия к сумме его составных частей.

В качестве активного элемента метаграфовой модели используются метаграфовые агенты на основе продукционных правил. Условием правила является фрагмент метаграфа, действием правила является множество операций, выполняемых над метаграфом.

Правила метаграфового агента можно разделить на замкнутые и разомкнутые.

Разомкнутые правила не меняют в правой части правила фрагмент метаграфа, относящийся к левой части правила. Можно разделить входной и выходной фрагменты метаграфа. Данные правила являются аналогом шаблона, который порождает выходной метаграф на основе входного.

Замкнутые правила меняют в правой части правила фрагмент метаграфа, относящийся к левой части правила. Изменение метаграфа в правой части правил заставляет срабатывать левые части других правил. Но при этом некорректно разработанные замкнутые правила могут привести к заикливанию метаграфового агента.

Таким образом, метаграфовый агент может генерировать один метаграф на основе другого (с использованием разомкнутых правил) или модифицировать метаграф (с использованием замкнутых правил).

Метаграфовая модель данных изоморфна системе предикатов высших порядков. Все основные элементы метаграфовой модели, включая агентов, могут быть преобразованы в предикатное описание. Важным свойством предложенного описания является самоотображаемость, что позволяет агентам модифицировать как метаграфовые данные, так и структуру агентов нижнего уровня.

*Расширенный реферат научной статьи на английском языке:*

The article is devoted to the predicate representation of metagraph data model. Traditionally, predicate representation is used in logical programming languages. Currently, it is believed that the approach based on the predicate representation and predicate-based logical languages are obsolete.

However, classical predicate form is convenient enough for syntax representation, as it resembles a function call in most programming languages. The syntax of predicate form is

significantly less diverse than the syntax of functional programming languages, which simplifies learning of languages based on predicate form. However, the absence of named parameters should be attributed to the shortcomings of the classical predicate form.

The opinion that logical programming languages are obsolete is refuted, first of all, by the principle of Curry-Howard correspondence. This principle establishes a correspondence between type systems (which are used in functional languages) and the logical calculus (which are the basis of logical languages). Thus, it is possible to speak about the isomorphism of logical and functional programming paradigms. Therefore, the opinion about the superiority of one paradigm is incorrect.

The mechanism of backtracking used in the Prolog, provides a convenient way to query the knowledge base. The classical Prolog implements the closed-world assumption; however, modern logical languages such as AnsProlog implement the open-world assumption. The classical Prolog doesn't implement the higher-order predicate calculus; however, it is supported by modern logic languages such as Flora-2. The classical Prolog doesn't support forward chaining, as an extension we propose forward chaining on metagraph using metagraph agents. Prolog is used in modern artificial intelligence projects, such as GATE (Prolog is used for constructing syntax trees and sentence processing) and HypergraphDB (dialect TuProlog is used for hypergraph data model processing).

In this article metagraph data model is proposed. The basic elements of this model are vertex, edge, metavertex, metaedge. All the elements of a metagraph are annotated with attributes. The presence of metavertex's own attributes and relations with other vertices is an important feature of metagraph model. This corresponds to the principle of emergence, which is, giving the metavertex a new quality, the impossibility to separate metavertex to the sum of its parts. Metagraph agents based on production rules are used as the active element of metagraph data model. The rule condition is a fragment of the metagraph; the rule action is a set of operations performed on the metagraph.

Metagraph agent rules can be divided into closed and open.

The action of open rule doesn't change the fragment of a metagraph relating to the condition of the rule. The input and output fragments of the metagraph can be separated. The open rule is similar to the template that generates the output metagraph based on the input metagraph.

The action of closed rule changes the fragment of a metagraph relating to the condition of the rule. The change of the metagraph by the action of the rule cause to check the conditions of other rules. But incorrectly designed closed rules may cause an infinite loop of metagraph agent.

Thus, metagraph agent can generate a metagraph based on another one (using open rules) or to modify the single metagraph (using closed rules).

The metagraph data model is isomorphic to the higher-order predicate model. The metagraph data model, including the agents, can be converted to a predicate representation. An important property of the proposed representation is homoiconicity, which means that agents can modify metagraph data model as well as the structure of the lower level agents.

*Ключевые слова:* предикатное описание, логическое программирование, метаграф, гиперграф, метавершина, метаребро, метаграфовый агент;

*Keywords:* predicate representation, logical programming, metagraph, hypergraph, metavertex, metaedge, metagraph agent.

## **1. Введение**

В настоящее время модели на основе сложных сетей находят все более широкое применение в различных областях технических и естественных наук. Сложные сети рассматриваются в работах И.А. Евина [1], О.П. Кузнецова и Л.Ю. Жиликовой [2], К.В. Анохина [3] и других исследователей.

На кафедре «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана в рамках данного направления предложена метаграфовая модель. Данную модель предлагается

применять как средство для описания сложных сетей [4], как средство для описания семантики и прагматики информационных систем [5], как средство для описания гибридных интеллектуальных информационных систем [6].

Однако в указанных работах не было предложено формализма для языкового описания метаграфовой модели. В данной статье предлагается использовать предикатное описание метаграфовой модели данных, что является основой для разработки метаграфового языка программирования. С точки зрения авторов, такой язык должен основываться на использовании логической парадигмы программирования. Большинство языков логического программирования используют в качестве основы предикатное описание.

В настоящее время существует мнение, что подход на основе предикатного описания и логические языки на основе этого подхода являются устаревшими. Данное мнение, как правило, не встречается в серьезных научных работах, но является доминирующим в сообществе разработчиков, активно обсуждается на форумах и в блогах.

Поэтому перед тем, как рассмотреть суть предлагаемой модели, нам кажется важным остановиться на деталях критики предикатного описания и логических языков. Основным объектом для критики, как правило, является язык логического программирования Пролог. Несмотря на то, что данная критика отчасти справедлива, логические языки программирования, в том числе на основе предикатного описания, продолжают развиваться, а многие критические замечания уже исправлены в современных системах.

## **2. Анализ критики логического программирования и языков на основе предикатного описания**

### **2.1. Критика языков на основе предикатного описания.**

Классическая предикатная форма синтаксиса языка содержит предикаты в виде «ПРЕДИКАТ(ПАРАМЕТРЫ)». Параметры представляются в виде списка с разделителями. Как правило, любой предикатный язык программирования содержит также дополнительные

конструкции, которые делают язык более выразительным, но в предикатных языках таких конструкций обычно немного.

В классическом Прологе используются неименованные параметры, которые задаются только позицией в списке параметров.

S-выражения языка LISP можно считать разновидностью предикатной формы, но в этом случае все выражение записывается в круглых скобках, и предикат синтаксически находится на месте первого параметра «(ПРЕДИКАТ ПАРАМЕТРЫ)». Такой вариант синтаксиса, наверное, можно считать менее удобным, так как предикат выделяется в синтаксисе менее явно. В некоторых диалектах LISP существует вариант именованных параметров, в котором имя параметра выделяется специальным символом, чаще всего двоеточием, например, «(ПРЕДИКАТ :ИМЯ1 ЗНАЧЕНИЕ1 :ИМЯ2 ЗНАЧЕНИЕ2)». Однако данный синтаксис также трудно признать удобным, потому что пары имя-значение не выделяются явно, что создает сложности при визуальном анализе длинных S-выражений.

В функциональных языках программирования каждый аспект языка, как правило, обозначается отдельной синтаксической конструкцией, и эти конструкции слабо унифицированы. Особенной можно считать ситуацию в языке Haskell, где используются расширения компилятора, которые позволяют добавлять в язык новые синтаксические конструкции, что делает синтаксис языка принципиально изменяемым. Это чрезвычайно усложняет освоение функциональных языков по сравнению с языками на основе предикатной модели.

С нашей точки зрения, классическая предикатная форма является достаточно удобной для описания, так как напоминает вызов функций в большинстве языков программирования. Такая форма выигрывает по выразительности у S-выражений языка LISP. Синтаксис предикатной формы существенно менее разнообразен, чем синтаксис функциональных языков, что упрощает изучение языков на основе предикатной формы. Однако отсутствие именованных параметров следует отнести к недостаткам классической предикатной формы.



## 2. Критика логической парадигмы программирования.

В рамках критики логической парадигмы программирования можно выделить следующие основные пункты.

2.1. Функциональные языки программирования с присущим им характерным синтаксисом – «современные», а логические языки на основе предикатной модели «устарели».

Это мнение особенно доминирует среди сторонников функционального программирования. Действительно, современные чистые функциональные (Haskell, Erlang) и объектно-функциональные (Scala, F#, OCaml) языки программирования не используют или ограниченно используют предикатное описание.

Однако существует современный язык функционально-логического программирования Mercury, который построен на основе предикатной модели. Анализ Mercury и Haskell [7] показывает, что языки в целом являются сравнимыми по своим возможностям.

Также необходимо отметить принцип изоморфизма Карри-Ховарда, который устанавливает соответствие между системами типов (которые применяются в функциональных языках) и логическими исчислениями (которые являются основой логических языков) [8]. Этот принцип утверждает, что для системы типов можно подобрать изоморфное логическое исчисление, и наоборот. В частности, известен изоморфизм логики высказываний и простого типизированного  $\lambda$ -исчисления. В соответствии с этим принципом некорректно говорить о «превосходстве» логической или функциональной парадигмы, так как между ними можно найти соответствие.

2.2. Механизм поиска с возвратом (backtracking), применяемый в Прологе, не нужен в «современных» языках.

Для подтверждения этого тезиса обычно приводят в пример язык Erlang, который построен на основе значительно переработанного синтаксиса Пролога, но переведен из

логической парадигмы в функциональную, механизм поиска с возвратом в нем не используется.

Однако механизм поиска с возвратом является основой обратного логического вывода в Прологе и фактически превращает Пролог в очень компактный язык запросов к данным.

Проблемой данного механизма может явиться большой объем обрабатываемой базы знаний. Для решения данной проблемы нами предлагается делить базу знаний на отдельные области-метаграфы и осуществлять их параллельную обработку с помощью многоагентного подхода.

### 2.3. Пролог реализует предположение о замкнутости мира.

Предположение о замкнутости мира (closed-world assumption) используется для упрощения логического вывода и позволяет считать ложными все утверждения, которые не являются истинными, даже если их ложность нельзя доказать, то есть понятия «нет» и «не знаю» считаются эквивалентными. В противоположность этому предположению используется предположение об открытости мира (open-world assumption), в котором явно разделяются все три понятия: «да», «нет» и «не знаю».

Классический Пролог действительно реализует предположение о замкнутости мира.

Однако существуют современные логические языки, которые используют предположение об открытости мира, в частности AnsProlog, который использует парадигму Answer Set Programming [9]. Но необходимо отметить, что написание правил вывода на этом языке намного сложнее, чем в классическом Прологе, поэтому некоторые современные языки (в том числе Mercury) не отказываются от предположения о замкнутости мира.

### 2.4. Пролог не поддерживает исчисление предикатов высших порядков.

Классический Пролог действительно не поддерживает исчисление предикатов высших порядков. Однако более современные языки его поддерживают, в частности, такой подход используется в проекте Flora-2 [10].

Также логика высших порядков поддерживается в проекте Teujus [11], но используемый язык программирования не использует предикатный синтаксис, и с синтаксической точки зрения ближе к функциональным языкам программирования.

#### 2.5. Пролог не поддерживает прямой логический вывод.

Классический Пролог действительно не поддерживает прямой логический вывод, так как является системой обратного логического вывода. В современных версиях Пролога элементы прямого логического вывода поддерживаются с использованием подхода на основе программирования в ограничениях (Constraint Logic Programming) [12].

Далее нами предлагается использование прямого логического вывода на метаграфах с использованием метаграфовых агентов.

#### 2.6. Пролог не используется в современных проектах, связанных с искусственным интеллектом.

Даже классический Пролог, несмотря на присущие ему ограничения, используется современных в проектах, связанных с искусственным интеллектом.

Первым примером является проект GATE, предназначенный для обработки естественного языка. В этом проекте язык Пролог применяется в подпроекте «SUPPLE Parser», с использованием которого реализовано конструирование синтаксических деревьев и обработка предложений [13].

Вторым примером является СУБД HypergraphDB. HypergraphDB является частью проекта OpenCog, предназначенного для разработки приложений в области искусственного интеллекта. HypergraphDB является графовой СУБД, которая использует модель данных в форме гиперграфа.

СУБД HypergraphDB интегрирована с диалектом Пролога TuProlog. Интеграция реализована таким образом, что данные, хранящиеся в СУБД, автоматически являются базой знаний программы на TuProlog, с помощью TuProlog можно проводить обработку данных в БД [14].

В целом данный подход нельзя назвать новым. Обработка данных в реляционных СУБД с помощью диалектов Пролога известна с 70-х годов прошлого века, для этой цели существует Пролог-подобный язык запросов Datalog. Но HypergraphDB является современным проектом и использует сложную графовую модель данных.

Подводя итог анализа критики, можно сделать следующие выводы:

- Предикатная форма синтаксиса является достаточно удобной и используется в ряде современных языков программирования (Mercury и другие современные диалекты Пролога).
- Логическую парадигму программирования нельзя признать устаревшей, так как она изоморфна функциональной парадигме. Многие недостатки классического Пролога исправлены в современных логических языках.

Далее рассмотрим предлагаемую метаграфовую модель данных и ее реализацию на основе предикатного описания.

### 3. Метаграфовая модель данных

В настоящее время одним из важных направлений в искусственном интеллекте является представление знаний с помощью сложных сетевых моделей. Например, в HypergraphDB в качестве такой модели используются гиперграфы.

В нашей работе [4] показано, что, хотя гиперграф и содержит гиперребра, но не позволяет моделировать сложные иерархические зависимости, и не является полноценной сложной «сетью с эмерджентностью». В качестве альтернативы гиперграфовой модели нами предлагается использовать метаграфовую модель, описание которой рассмотрено в данном разделе:

$$MG = \langle V, MV, E, ME \rangle,$$

где MG – метаграф; V – множество вершин метаграфа; MV – множество метавершин метаграфа; E – множество ребер метаграфа; ME – множество метаребер метаграфа.

Вершина метаграфа характеризуется множеством атрибутов:

$$v_i = \{atr_k\}, v_i \in V,$$

где  $v_i$  – вершина метаграфа;  $atr_k$  – атрибут.

Ребро метаграфа характеризуется множеством атрибутов, исходной и конечной вершиной и признаком направленности:

$$e_i = \langle v_s, v_e, eo, \{atr_k\} \rangle, e_i \in E, eo = true \mid false,$$

где  $e_i$  – ребро метаграфа;  $v_s$  – исходная вершина (метавершина) ребра;  $v_e$  – конечная вершина (метавершина) ребра;  $eo$  – признак направленности ребра ( $eo=true$  – направленное ребро,  $eo=false$  – ненаправленное ребро);  $atr_k$  – атрибут.

Фрагмент метаграфа:

$$MG_i = \{ev_j\}, ev_j \in (V \cup E \cup MV \cup ME),$$

где  $MG_i$  – фрагмент метаграфа;  $ev_j$  – элемент, принадлежащий объединению множеств вершин (метавершин) и ребер (метаребер) метаграфа.

Таким образом, фрагмент метаграфа в общем виде может содержать произвольные вершины (метавершины) и ребра (метаребра) без ограничений. Ограничения вводятся на фрагменты метаграфа, входящие в метавершину и метаребро.

Метавершина метаграфа:

$$mv_i = \langle \{atr_k\}, \{ev_j\} \rangle, mv_i \in MV, ev_j \in (V \cup E^{eo=false} \cup MV \cup ME^{eo=false}),$$

где  $mv_i$  – вершина метаграфа;  $atr_k$  – атрибут,  $ev_j$  – элемент, принадлежащий объединению множеств вершин (метавершин) и ребер (метаребер) метаграфа.

Таким образом, метавершина, в дополнение к свойствам вершины, включает вложенный фрагмент метаграфа. При этом ребра и метаребра этого фрагмента могут быть только ненаправленными,  $eo=false$ .

Метаребро метаграфа:

$$me_i = \langle v_s, v_e, eo, \{atr_k\}, \{ev_j\} \rangle, e_i \in E, eo = true \mid false,$$

$$ev_j \in (V \cup E^{eo=true} \cup MV \cup ME^{eo=true}),$$

где  $me_i$  – метаребро метаграфа;  $v_s$  – исходная вершина (метавершина) ребра;  $v_E$  – конечная вершина (метавершина) ребра;  $eo$  – признак направленности метаребра ( $eo=true$  – направленное метаребро,  $eo=false$  – ненаправленное метаребро);  $atr_k$  – атрибут;  $ev_j$  – элемент, принадлежащий объединению множеств вершин (метавершин) и ребер (метаребер) метаграфа.

Таким образом, метаребро, в дополнение к свойствам ребра, включает вложенный фрагмент метаграфа. При этом ребра и метаребра этого фрагмента могут быть только направленными,  $eo=true$ .

Определения метавершины и метаребра являются рекурсивными, так как элементы  $ev_j$  могут быть, в свою очередь, метавершинами и метаребрами.

Наличие у метавершин собственных атрибутов и связей с другими вершинами является важной особенностью метаграфов. Это соответствует принципу эмерджентности, то есть приданию понятию нового качества, несводимости понятия к сумме его составных частей. Фактически, как только вводится новое понятие в виде метавершины, оно «получает право» на собственные свойства, связи и т.д., так как, в соответствии с принципом эмерджентности, новое понятие обладает новым качеством и не может быть сведено к подграфу базовых понятий.

В качестве активного элемента метаграфовой модели используются метаграфовые агенты. Определим метаграфовый агент следующим образом:

$$ag^M = \langle MG_D, R, AG^{ST} \rangle, R = \{r_j\}$$

где  $ag^M$  – метаграфовый агент;  $MG_D$  – фрагмент метаграфа, на основе которого выполняются правила агента (рабочий метаграф);  $R$  – набор правил (множество правил  $r_j$ );  $AG^{ST}$  – стартовое условие выполнения агента (фрагмент метаграфа, который используется для стартовой проверки правил, или стартовое правило).

Структура правила метаграфового агента:

$$r_i : MG_j \rightarrow OP^{MG},$$

где  $r_i$  – правило;  $MG_j$  – фрагмент метаграфа, на основе которого выполняется правило;  $OP^{MG}$  – множество операций, выполняемых над метаграфом.

Антецедентом (условием, левой частью) правила является фрагмент метаграфа, консеквентом (действием, правой частью) правила является множество операций, выполняемых над метаграфом.

Правила метаграфового агента можно разделить на замкнутые и разомкнутые.

Разомкнутые правила не меняют в правой части правила фрагмент метаграфа, относящийся к левой части правила. Можно разделить входной и выходной фрагменты метаграфа. Данные правила являются аналогом шаблона, который порождает выходной метаграф на основе входного.

Замкнутые правила меняют в правой части правила фрагмент метаграфа, относящийся к левой части правила. Изменение метаграфа в правой части правил заставляет срабатывать левые части других правил. Но при этом некорректно разработанные замкнутые правила могут привести к заикливанию метаграфового агента.

Таким образом, метаграфовый агент позволяет генерировать один метаграф на основе другого (с использованием разомкнутых правил) или модифицировать метаграф (с использованием замкнутых правил).

Отметим, что в классическом Прологе аналогом этих действий являются операторы `assert` и `retract`, которые позволяют динамически добавлять и удалять предикаты из базы знаний. При этом база знаний является единой, и при ее увеличении скорость логического вывода падает.

Использование многоагентного подхода позволяет превратить каждый фрагмент метаграфа в отдельную базу знаний, которая может быть динамически изменена. Наличие многоагентности позволяет производить параллельный логический вывод на различных фрагментах метаграфа. Использование подхода на основе продукционных правил позволяет

использовать прямой логический вывод в дополнение к обратному выводу, при этом обратный вывод используется в качестве средства запросов к данным.

Далее рассмотрим, как предложенная модель может быть представлена в виде предикатного описания.

#### **4. Предикатное описание метаграфовой модели данных**

Интересной особенностью метаграфовой модели является то, что она изоморфна системе предикатов высших порядков. Рассмотрим более подробно различные варианты отображения элементов метаграфовой модели в предикатное описание, которые представлены в таблице 1.

Метавершина изоморфна предикату. В таблице 1 (вариант 1) показан пример метавершины  $mv_1$ , который содержит три вложенных несвязанных вершины  $v_1$ ,  $v_2$  и  $v_3$ . Предикат соответствует метавершине, вершины изоморфны переменным, которые являются параметрами предиката. В качестве имени предиката используется соответствующий элемент метаграфовой модели (в случае метавершины «Metavertex»), имя метавершины задается именованным параметром Name. Данный случай является простейшим, поскольку вложенные вершины не связаны друг с другом, и метавершина в этом случае изоморфна гиперребру гиперграфа.

Ребро метаграфа можно рассматривать как частный случай метавершины, которая включает исходную и конечную вершину. Пример для ненаправленного ребра показан в виде варианта 2. В предикатном описании в этом случае ребро можно рассматривать как предикат, а исходную и конечную вершину как параметры предиката. Для ребра используется имя предиката «Edge».

Пример ненаправленного ребра, который полностью соответствует формальному определению, показан в виде варианта 3. В этом случае метавершина помечается соответствующей аннотацией направленности, а в предикат добавляется именованный параметр, соответствующий признаку направленности.



Пример направленного ребра показан в виде варианта 4. В предлагаемой предикатной модели все параметры могут быть именованными. В варианте 4 (вариант предикатного представления 2) именованные параметры соответствуют параметрам из формального определения ребра метаграфа.

Метавершина, содержащая вершины и ребра, может быть представлена с использованием предикатов высших порядков. Вариант 5 содержит пример метавершины с ненаправленными ребрами, а вариант 6 – пример метавершины с направленными ребрами. В соответствии с определением метаграфа предикаты, отвечающие за вершины и ребра, вложены на одном уровне в предикат метавершины.

Метаребро метаграфа также может быть представлено с помощью предикатов высших порядков. Вариант 7 содержит пример метаребра  $me_1$ , исходной вершиной которого является вершина  $v_2$ , конечной метавершиной  $mv_3$ , при этом метаребро содержит вложенную метавершину  $mv_4$  (детальное содержимое вершины и метавершин не показано, чтобы не загромождать пример). Для метаребра используется имя предиката «Metaedge».

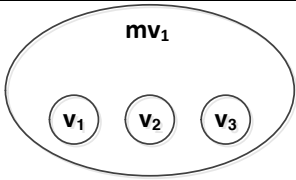
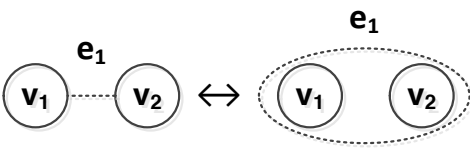
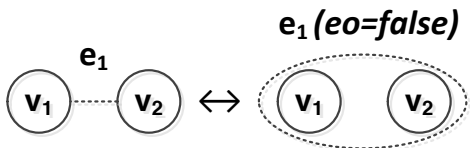
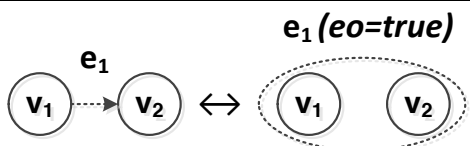
Фрагмент метаграфа может содержать произвольное количество вершин (метавершин) и ребер (метаребер) метаграфа. Вариант 8 содержит пример фрагмента метаграфа  $mg_0$ , содержащего вершину  $v_2$ , метавершины  $mv_3$  и  $mv_5$ , метаребро  $me_1$ . Для фрагмента метаграфа используется имя предиката «Metagraph», для вершины имя предиката «Vertex».

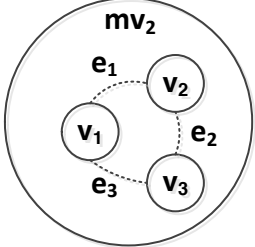
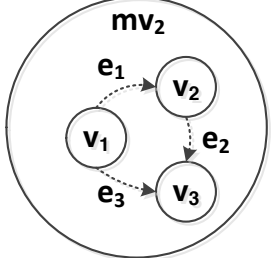
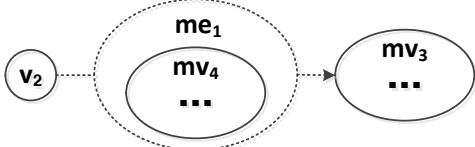
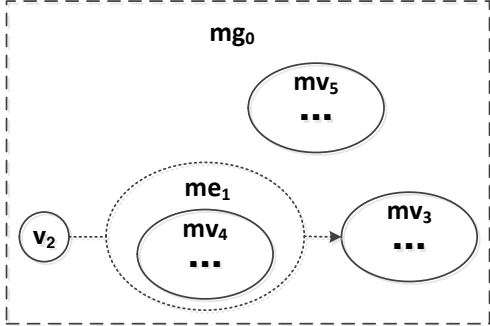
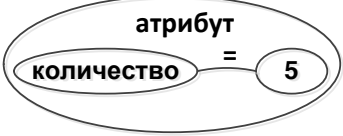
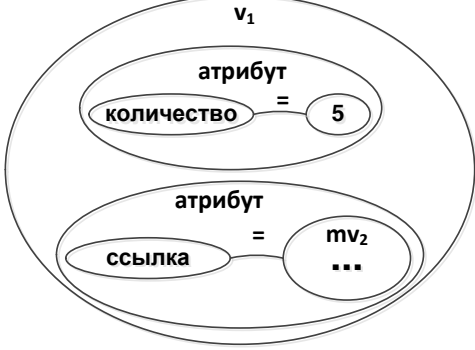
Атрибут может быть представлен как частный случай метавершины, содержащий имя и значение. Вариант 9 показывает пример атрибута, содержащего целое число. Вариант 10 показывает пример вершины  $v_1$ , содержащей атрибут из примера 9, а также атрибут, ссылающийся на метавершину  $mv_2$ . Для атрибута используется имя предиката «Attribute».

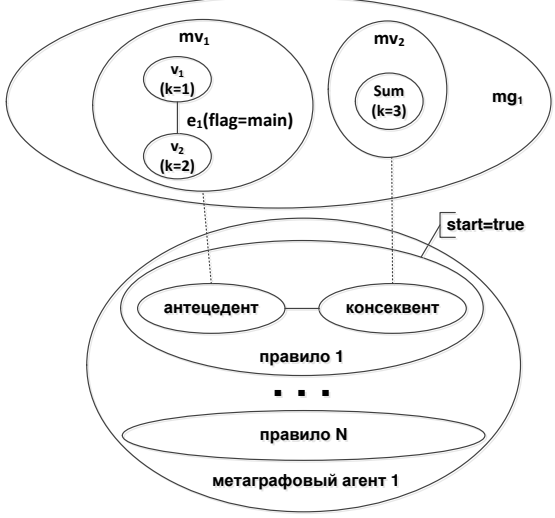
Метаграфовый агент также может быть представлен в виде предикатного описания. Вариант 11 показывает пример метаграфового агента с именем «метаграфовый агент 1» (предикат «Agent»). Рабочим метаграфом является  $mg_1$  (параметр «WorkMetagraph»). Описание правил содержит предикат «Rules». Правилу соответствует предикат «Rule».

Стартовым правилом (параметр «start=true» предиката «Rule») является «правило 1» (остальные правила не показаны, чтобы не загромождать пример). Условию правила соответствует предикат «Condition». Параметр «WorkMetagraph» содержит ссылку на проверяемую метавершину  $mv_1$ . Условие проверяет, что метавершина  $mv_1$  содержит вершины  $v_1$  и  $v_2$ , содержащие атрибуты «k». Найденные значения атрибутов «k» помещаются в переменные \$k1 и \$k2. Вершины  $v_1$  и  $v_2$  должны быть соединены ребром, содержащим атрибут «flag=main». Если условие выполняется, и найденный фрагмент метаграфа найден, то выполняется действие правила, которому соответствует предикат «Action». Параметр «WorkMetagraph» содержит ссылку на результирующую метавершину  $mv_2$ . В действии выполняется добавление новых элементов (предикат «Add»). Добавляется вершина Sum, содержащая атрибут «k=\$k1+\$k2». Для обозначения вычисляемого выражения используется предикат «Eval».

Таблица 1. Соответствие фрагментов метаграфовой модели предикатному описанию.

Вариант	Фрагмент метаграфа	Предикатное представление
1		Metavertex(Name= $mv_1$ , $v_1$ , $v_2$ , $v_3$ )
2		Edge(Name= $e_1$ , $v_1$ , $v_2$ )
3		Edge(Name= $e_1$ , $v_1$ , $v_2$ , eo=false)
4		1. Edge(Name= $e_1$ , $v_1$ , $v_2$ , eo=true) 2. Edge(Name= $e_1$ , $v_S=v_1$ , $v_E=v_2$ , eo=true)

5		Metavertex(Name=mv2, v1, v2, v3, Edge (Name=e1, v1, v2), Edge(Name=e2, v2, v3), Edge(Name=e3, v1, v3))
6		Metavertex(Name=mv2, v1, v2, v3, Edge(Name=e1, vS=v1, vE=v2, eo=true), Edge(Name=e2, vS=v2, vE=v3, eo=true), Edge(Name=e3, vS=v1, vE=v3, eo=true))
7		Metaedge(Name=me1, vS=v2, vE=mv3, Metavertex(Name=mv4, ...), eo=true)
8		Metagraph(Name=mg0, Vertex(Name=v2, ...), Metavertex(Name=mv3, ...), Metavertex(Name=mv5, ...), Metaedge(Name=me1, vS=v2, vE=mv3, Metavertex(Name=mv4, ...), eo=true))
9		Attribute(количество, 5)
10		Vertex(Name=v1, Attribute(количество, 5), Attribute(ссылка, mv2))

11		<p>Agent(Name='метаграфовый агент 1',  WorkMetagraph=mg<sub>1</sub>, Rules(  Rule(Name='правило 1', start=true,  Condition(WorkMetagraph=mv<sub>1</sub>,  Vertex(Name=v<sub>1</sub>, Attribute(k, \$k1)),  Vertex(Name=v<sub>2</sub>, Attribute(k, \$k2)),  Edge(v<sub>1</sub>, v<sub>2</sub>, Attribute(flag, main)))  Action(WorkMetagraph=mv<sub>2</sub>,  Add(Vertex(Name=Sum,  Attribute(k, Eval(\$k1+\$k2))))))  ), Rule(...) ... ))</p>
----	---	--

Таким образом, сформировано предикатное описание всех основных элементов метаграфовой модели данных.

Предложенное предикатное описание является основой языка обработки метаграфовой модели данных, который включает описание как элементов метаграфовой модели данных, так и описание средств обработки метаграфов – метаграфовых агентов.

Отметим, что предложенное описание обладает свойством самоотображаемости. Самоотображаемость (англ. homoiconicity) – это способность языка программирования анализировать программу на этом языке как структуру данных.

Поскольку и для описания данных, и для описания агентов используется предикатный подход, то предложенный способ позволяет агентам верхнего уровня модифицировать структуру агентов нижнего уровня на основе модификации предикатных описаний. В соответствии с [6] данная особенность является одним из необходимых условий разработки гибридных интеллектуальных информационных систем.

## 5. Заключение

В статье предложен подход к использованию предикатного описания для метаграфовой модели данных.

Предикатная форма описания синтаксиса языка является достаточно удобной и используется в ряде современных языков программирования. Синтаксис предикатной формы существенно менее разнообразен, чем синтаксис функциональных языков, что упрощает изучение языков на основе предикатной формы. Однако отсутствие именованных параметров следует отнести к недостаткам классической предикатной формы.

Логическую парадигму программирования в целом нельзя признать устаревшей, так как она изоморфна функциональной парадигме. Многие недостатки классического Пролога исправлены в современных логических языках. Но необходимо отметить, что в настоящее время функциональные языки развиваются значительно активнее логических.

Метаграфовая модель данных содержит описание вершин, метавершин, ребер и метаребер. Активным элементом метаграфовой модели является метаграфовый агент.

Метаграфовая модель данных, включая агентов, может быть представлена в форме предикатного описания, которое обладает свойством самоотображаемости.

Предикатное описание данных и агентов позволяет агентам верхнего уровня модифицировать структуру агентов нижнего уровня, что является важной особенностью разработки гибридных интеллектуальных информационных систем.

## **Список литературы**

1. Евин И.А. Введение с теорию сложных сетей //Компьютерные исследования и моделирование. 2010, Том 2, №2, с. 121-141.
2. Кузнецов О.П., Жиликова Л.Ю. Сложные сети и когнитивные науки // Нейроинформатика-2015. XVII Всероссийская научно-техническая конференция. Сборник научных трудов. Ч. 1. М.: МИФИ. 2015. С. 18.
3. Анохин К.В. Когнитом: гиперсетевая модель мозга // Нейроинформатика-2015. XVII Всероссийская научно-техническая конференция. Сборник научных трудов. Ч. 1. М.: НИЯУ МИФИ. 2015. С. 14-15.

4. Черненький В.М., Терехов В.И., Гапанюк Ю.Е. Представление сложных сетей на основе метаграфов // Нейроинформатика-2016. XVIII Всероссийская научно-техническая конференция. Сборник научных трудов. Ч. 1. М.: НИЯУ МИФИ, 2016.
5. Самохвалов Э.Н., Ревунков Г.И., Гапанюк Ю.Е. Использование метаграфов для описания семантики и прагматики информационных систем. Вестник МГТУ им. Н.Э. Баумана. Сер. «Приборостроение». 2015. Выпуск №1. С. 83-99.
6. Черненький В.М., Терехов В.И., Гапанюк Ю.Е. Структура гибридной интеллектуальной информационной системы на основе метаграфов. Нейрокомпьютеры: разработка, применение. 2016. Выпуск №9. С. 3-14.
7. Comparing Mercury with Haskell. Режим доступа: [https://www.mercurylang.org/about/comparison\\_with\\_haskell.html](https://www.mercurylang.org/about/comparison_with_haskell.html) (дата обращения 20.10.2016).
8. Пирс Б. Типы в языках программирования. – Добросвет, 2012. – 680 с.
9. Gelfond M., Kahl Yu. Knowledge representation, reasoning, and the design of intelligent agents : the answer-set programming approach. Cambridge University Press, 2014. – 366 p.
10. Flora-2 User's Manual. Version 1.1. September 5, 2015. Режим доступа: <http://flora.sourceforge.net/docs/floraManual.pdf> (дата обращения 20.10.2016).
11. Miller D., Nadathur G. Programming with Higher-Order Logic. Cambridge University Press, 2012. – 322 p.
12. Bratko I. Prolog Programming for Artificial Intelligence. Fourth Edition. Pearson Education Limited, 2012. – 697 p.
13. SUPPLE Parser. Режим доступа: <https://gate.ac.uk/sale/tao/splitch18.html#x23-43700018.3> (дата обращения 20.10.2016).
14. TuProlog Interpreter Integration With HyperGraphDB. Режим доступа: <https://github.com/hypergraphdb/hypergraphdb/wiki/TuProlog> (дата обращения 20.10.2016).

**Сведения об авторах**

**Гапанюк Юрий Евгеньевич**

*Год рождения:* 1974

*Год окончания вуза и его название:* 1998, МГТУ им. Н.Э. Баумана

*Ученая степень:* к.т.н.

*Место работы, должность:* доцент кафедры ИУ-5 МГТУ им. Н.Э. Баумана

*Полный адрес организации:* Московский Государственный Технический Университет им.

Н.Э. Баумана, г. Москва, 2-я Бауманская ул., д. 5, стр. 1, почтовый индекс: 105005

*Количество опубликованных работ:* около 20

*Область научных интересов:* проектирование автоматизированных систем

*Электронная почта:* gapyu@bmstu.ru

*Контактный телефон:* 8 916 558 94 30

**Ревунков Георгий Иванович**

*Год рождения:* 1948

*Год окончания вуза и его название:* 1971, МВТУ им. Н.Э. Баумана

*Ученая степень:* к.т.н.

*Место работы, должность:* доцент кафедры ИУ-5 МГТУ им. Н.Э. Баумана

*Полный адрес организации:* Московский Государственный Технический Университет им.

Н.Э. Баумана, г. Москва, 2-я Бауманская ул., д. 5, стр. 1, почтовый индекс: 105005

*Количество опубликованных работ:* более 50

*Область научных интересов:* базы данных, проектирование автоматизированных систем

*Электронная почта:* revunkov@bmstu.ru

*Контактный телефон:* 8 499 263 62 16

**Федоренко Юрий Сергеевич**

*Год рождения:* 1992

*Год окончания вуза и его название:* 2015, МГТУ им. Н.Э. Баумана

*Ученая степень:* магистр

*Место работы, должность:* аспирант кафедры ИУ-5 МГТУ им. Н.Э. Баумана

*Полный адрес организации:* Московский Государственный Технический Университет им. Н.Э. Баумана, г. Москва, 2-я Бауманская ул., д. 5, стр. 1, почтовый индекс: 105005

*Количество опубликованных работ:* около 10

*Область научных интересов:* проектирование интеллектуальных систем

*Электронная почта:* fedyura1992@yandex.ru

*Контактный телефон:* 8 915 477 13 45

### **Gapanyuk Yuriy Evgenievich**

*Год рождения:* 1974

*Год окончания вуза и его название:* 1998, Bauman Moscow State Technical University

*Ученая степень:* Ph.D. (Computer Sciences)

*Место работы, должность:* associate professor of Information Processing and Control Systems Department at Bauman Moscow State Technical University

*Полный адрес организации:* Bauman Moscow State Technical University, ul. Baumanskaya 2-ya, 5, Moscow, postcode: 105005

*Количество опубликованных работ:* about 20 publications

*Область научных интересов:* designing of automated systems

*Электронная почта:* gapyu@bmstu.ru

*Контактный телефон:* 8 916 558 94 30

### **Revunkov Georgiy Ivanovich**

*Год рождения:* 1948

*Год окончания вуза и его название:* 1971, Moscow Higher Technical School Named After N.E. Bauman

*Ученая степень:* Ph.D. (Engineering)



*Место работы, должность:* associate professor of Computer Science and Control Systems Department at Bauman Moscow State Technical University

*Полный адрес организации:* Bauman Moscow State Technical University, ul. Baumanskaya 2-ya, 5, Moscow, postcode: 105005

*Количество опубликованных работ:* more then 50 publications

*Область научных интересов:* databases, designing of automated systems

*Электронная почта:* revunkov@bmstu.ru

*Контактный телефон:* 8 499 263 62 16

### **Fedorenko Yuriy Sergeevich**

*Год рождения:* 1992

*Год окончания вуза и его название:* 2015, Bauman Moscow State Technical University

*Ученая степень:* Master (Computer Sciences)

*Место работы, должность:* postgraduate student of Information Processing and Control Systems Department at Bauman Moscow State Technical University

*Полный адрес организации:* Bauman Moscow State Technical University, ul. Baumanskaya 2-ya, 5, Moscow, postcode: 105005

*Количество опубликованных работ:* about 10 publications

*Область научных интересов:* designing of intelligent systems

*Электронная почта:* fedyura1992@yandex.ru

*Контактный телефон:* 8 915 477 13 45