Таким образом, в результате работы были предложены математические методы для оценки динамики изменения значений числовых показателей, основанные на методе кластерного анализа и теории нечетких множеств. Также были разработаны методы для качественной оценки значений числовых показателей, опирающиеся на теорию нечетких множеств. Кроме того, предложены различные виды функций устаревания значений показателей и различные варианты расчета коэффициентов устаревания кластеров значений показателей.

СПИСОК ЛИТЕРАТУРЫ

1. Уэно X., Исудзука М. Представление и использование знаний / М.: Мир, 1989. С. 50-54.
2. Джексон П. Введение в экспертные системы. М.: Вильямс, 2000. С. 213-217.
3. Кобринский Б.А., Фельдман А.Е. Анализ и учет ассоциативных знаний в медицинских экспертных системах //Новости искусств. интеллекта. 1995. № 3. С. 90-96.
4. Нечеткие множества в моделях управления и искусственного интеллекта / Под ред. Д.А. Поспелова. М.: Наука, 1986. 312 с.

***Александр Григорьевич ИВАШКО —***
***зав. кафедрой информационных систем***
***Института математики и компьютерных наук,***
***доктор технических наук, профессор***

***Михаил Викторович ГРИГОРЬЕВ —***
***старший преподаватель***
***кафедры информационных систем***
***Института математики и компьютерных наук***

***Тюменский государственный университет***

**УДК 004.4'41**

# *ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ЯЗЫК ОГРАНИЧЕНИЙ ДЛЯ ВЕРИФИКАЦИИ ПРОЦЕССА КОМАНДНОЙ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ*

*АННОТАЦИЯ. В статье описывается компилятор формального языка, реализующий правила верификации процесса разработки программного обеспечения в команде.*

*In article implementation of the compiler of the created formal description language of rules of verification of process of the software development in the command is described.*

Современные методы программной инженерии предполагают итеративную командную разработку. Данный подход позволяет в короткие сроки создавать качественные программные продукты. Однако допущение ошибки одним из участников проекта на любом этапе может привести к созданию новой итерации, что в свою очередь потребует дополнительных ресурсов. В связи с этим задача верификации процессов выполнения программного проекта становится крайне важной. Многие из правил, известные в литературе, могут использоваться в настоящее время только при верификации в ручном режиме, что

требует больших временных затрат. Нами предлагается способ верификации, который основывается на динамической проверке соответствия артефактов проекта созданному набору правил.

Реализация динамической верификации процессов инжиниринга программного обеспечения требует построения формализованной модели разработки, которая могла бы «работать» с артефактами данного процесса. В силу того, что язык Unified Modeling Language (UML) [1] стал стандартом при построении большинства артефактов программной инженерии, формализованная модель должна быть определена применением объектно-ориентированного подхода в спецификации Software Process Engineering Metamodel (SPEM) [2].

Правила, позволяющие выверить процесс разработки, следует применять как к артефактам проекта, определяя тем самым их трассируемость, так и к структуре процесса, описывающей семантику взаимодействия артефактов. Объектно-ориентированный подход к представлению структуры процесса и артефактов предполагает использование элементов этих представлений при описании соответствующих правил. Структура процесса SPEM и моделей UML определена при помощи метаобъектного представления спецификации Meta Object Facility (MOF) [3], предлагающей обобщенную структуру объектной модели:

$$M = (\mathrm{CLASS},\ \mathrm{ATT}_C,\ \mathrm{OP}_C,\ \mathrm{ASSOC},\ \mathrm{associates},\ \mathrm{roles},\ \mathrm{multiplicities}, \prec), \qquad (1)$$

где:

CLASS — множество классов, класс $c \in \mathrm{CLASS}$;

$\mathrm{ATT}_C$ — множество сигнатур операций для функций отображения значений полей, ассоциированных объекту класса $c$;

$\mathrm{OP}_C$ — множество сигнатур для пользовательских методов класса $c$;

ASSOC — множество наименований ассоциаций;

associates — функция отображения каждого наименования ассоциации на список участвующих классов;

roles — функция назначения каждому концу ассоциации наименования роли,

multiplicities — функция назначения каждому концу ассоциации порядка множественности,

$\prec$ — отношение частичного порядка на множестве CLASS, отражающее обобщенную иерархию классов.

Объектная модель (1) использует следующее описание класса.

Класс наследует поля, методы и ассоциации родительских классов:

$$\text{родительский класс: } \begin{cases} CLASS \to \mathrm{P}(CLASS) \\ c \to \{c' \mid c' \in CLASS \wedge c \prec c'\} \end{cases}. \qquad (2)$$

Полный набор полей класса с включает в себя все унаследованные поля и поля, определенные в самом классе:

$$ATT_C^{*} = ATT_C \cup \bigcup ATT_{C'}, c' \in \text{ родительский класс } (c). \qquad (3)$$

Родительский класс также может быть потомком другого класса, поэтому дочерний класс наследует атрибуты всех классов-предков, для чего используется оператор объединения.

Аналогично определяется набор методов:

$$OP_C^* = OP_C \cup \bigcup OP_{C'}, c' \in \text{родительский класс } (c) \qquad (4)$$

Наименование отношений между классами определяется следующим образом:

$$\text{navends}^*(c) = \text{navends}(c) \cup \bigcup \text{navends}(c'),$$
$$c' \in \text{родительский класс}(c) \qquad (5)$$

Таким образом, язык описания правил верификации должен использовать все элементы вышеописанной модели, обуславливая поддержку объектно-ориентированной архитектуры. Одним из таких языков является Object Constraint Language (OCL) [4], позволяющий описывать правила, применяемые к объектным моделям UML. Однако OCL представляет собой декларативный язык программирования, позволяющий определять дополнительную семантику объектной модели, что делает невозможным его использование для динамической автоматической верификации. Создание языка, основанного на OCL, т.е. позволяющего описывать правила, применимые к объектным моделям, при этом транслируемого в машинный код, предоставит возможность реализации механизма автоматизированной верификации процесса разработки.

Нами был разработан язык fOCL, в котором формализация правил, декларированных в OCL, дает возможность выполнять верификацию артефактов, поддерживающих объектную модель разработки, в автоматическом режиме.

Язык fOCL предоставляет контекст обращения к классам модели, используя именованные элементы (3), (4), (5) каждого класса. Разработанный язык содержит набор операторов, используя суперпозицию которых, разработчик может составлять выражения любой сложности. Декларативная мощность языка достигается использованием всех ключевых слов и функций OCL, что позволяет использовать все возможности объектного языка ограничений при написании выражений.

Разработка выражений на языке fOCL должна сопровождаться преобразованием исходного текста правил в исполняемый код (компиляцией). Набор правил может уточняться и дополняться, следовательно, ставится задача создания компилятора, позволяющего получать машинный код, эквивалентный исходным правилам.

Компилятор языка fOCL реализован в виде 4 модулей: лексический анализатор, синтаксический анализатор, семантический анализатор, генератор целевого кода. В силу того, что компилятор предназначен для генерации кода на основе узкоспециализированного языка, имеющего жесткую структуру и не позволяющего создания пользовательских процедур, нецелесообразна реализация модулей генерации промежуточного кода и оптимизатора кода [5].

На этапе лексического анализа происходит преобразование текста исходного выражения в набор лексем, при этом используется весь формальный алфавит языка fOCL:

$$V = K \cup S \cup O \cup F \cup T \cup CLASS \cup ATT \cup OP, \qquad (6)$$

где:

K={and, attr, context, inv, pre, post, package, body, init, derive, let, def, if, then, else, endif, endpackage, implies, in, not, oper, or, xor, self} — множество ключевых слов;

S={a..z, A..Z, 0..9, :, ',--, [, ]} — множество допустимых символов;

O={+, −, *, /, >, <, =, >=, <=, <>, (, ), .., −>, ∧} — множество операторов;

F={oclIsNew(), oclIsUndefined(), oclIsInvalid(), oclAsType(), oclIsTypeOf(), oclIsKindOf(), oclIsInState(), allInstances(),hasReturned(), result(), isSignalSent(), isOperationCall(), abs(), floor(), round(), max(), min(), div(), mod(), size(), concat(), substring(), toInteger(), toReal(), includes(), excludes(), count(), includesAll(), excludesAll(), isEmpty(), notEmpty(), sum(), product(), union(), intersection(), $\bar{n}()$, including(), excluding(), symmetricDifference(), flatten(), asSet(), asOrderedSet(), asSequence(), asBag(), append(), prepend(), insertAt(), subOrderedSet(), at(), indexOf(), first(), last(), subSequence(), exists(), forAll(), isUnique(), any(), one(), collect(), select(), reject(), collectNested(), sortedBy(), iterate()} — набор встроенных функций языка;

T={Boolean, Integer, Real, String, OclAny, OclVoid, OclInvalid, OclMessage, OclElement, OclType, UnlimitedInteger, Collection, Set, OrderedSet, Bag, Sequence, ExpressionInOcl} — множество, определяющее возможные типы, используемые в выражениях языка.

Полученная последовательность лексем передается синтаксическому анализатору, который в свою очередь проверяет, может ли данный поток быть порожденным грамматикой языка, используя дерево синтаксического разбора (рис 1).

На этапе семантического анализа компилятор обращается непосредственно к используемой объектной модели с целью сопоставления конструкций выражения элементам модели. При этом также осуществляется проверка взаимодействия классов исходной модели, используя отношение ассоциации.

Объектная модель, представленная в графической нотации UML, преобразуется в специальный формат обмена XML Metadata Interchange (XMI) [6], представляющий собой структурированный XML-файл, изоморфный исходной модели. Анализатор обращается к содержимому файла, извлекает информацию о классах, полях и методах каждого класса и дополняет дерево синтаксического разбора полученными элементами (рис. 2). При включении элементов в дерево производится проверка выражения действительным ассоциациям используемых классов.

Далее синтаксическое дерево проверяется на соответствие типам. Язык использует:

— набор стандартных типов (Integer, Real, Boolean, String) — для описания конструкций;

— дополнительные типы (OclVoid, OclElement, OclType, UnlimitedInteger, Collection, Set, OrderedSet, Bag, Sequence) — для реализации возможности взаимодействия с данными используемых моделей;

— специальные типы (OclAny, OclInvalid, OclMessage, ExpressionInOcl) — для реализации вывода сообщений компилятора (например, сообщений об ошибках компиляции).

Используя контекстно-свободную формальную грамматику языка [7], определяются типы данных для всех нетерминалов синтаксического дерева. Под нетерминалом понимается выражение языка, не имеющее конкретного символьного значения. Схема трансляции, описываемая в грамматике, позволяет определить тип для синтезируемых узлов дерева и порождаемых атрибутов. Все полученные идентификаторы исходного текста программы записываются в таблицу символов с указанием их типа. При несовпадении типов или невозможности их приведения выражению назначается тип OclInvalid, используемый для определения ошибки.

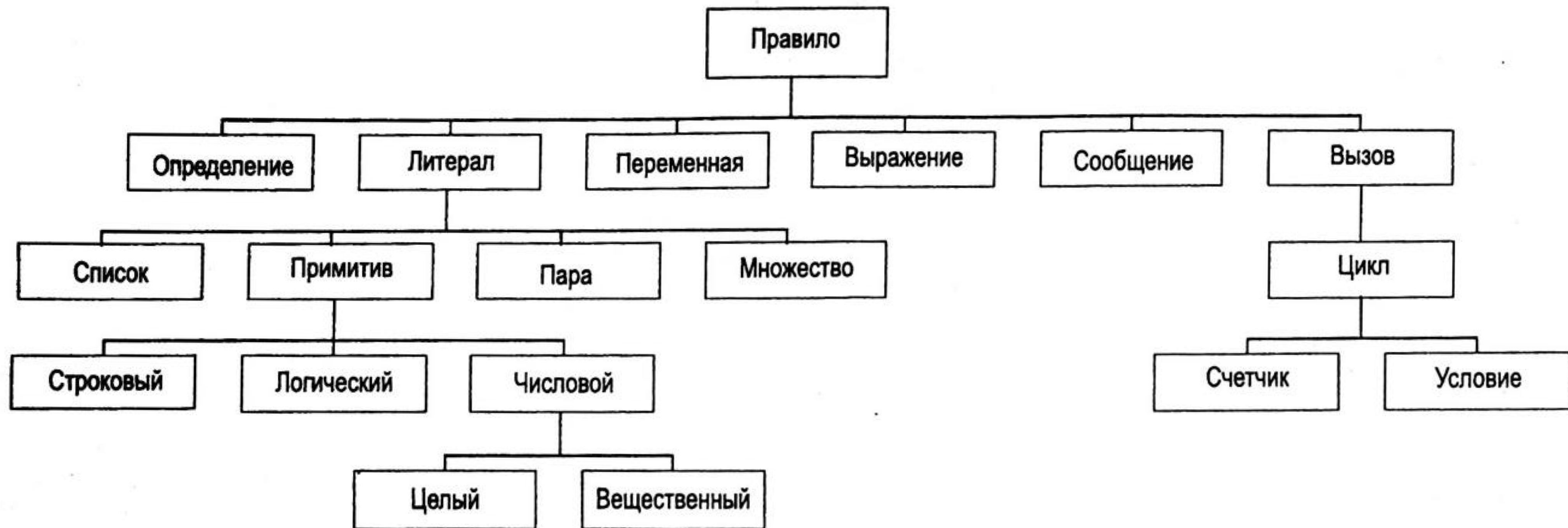


*Рис. 1.* Дерево синтаксического разбора

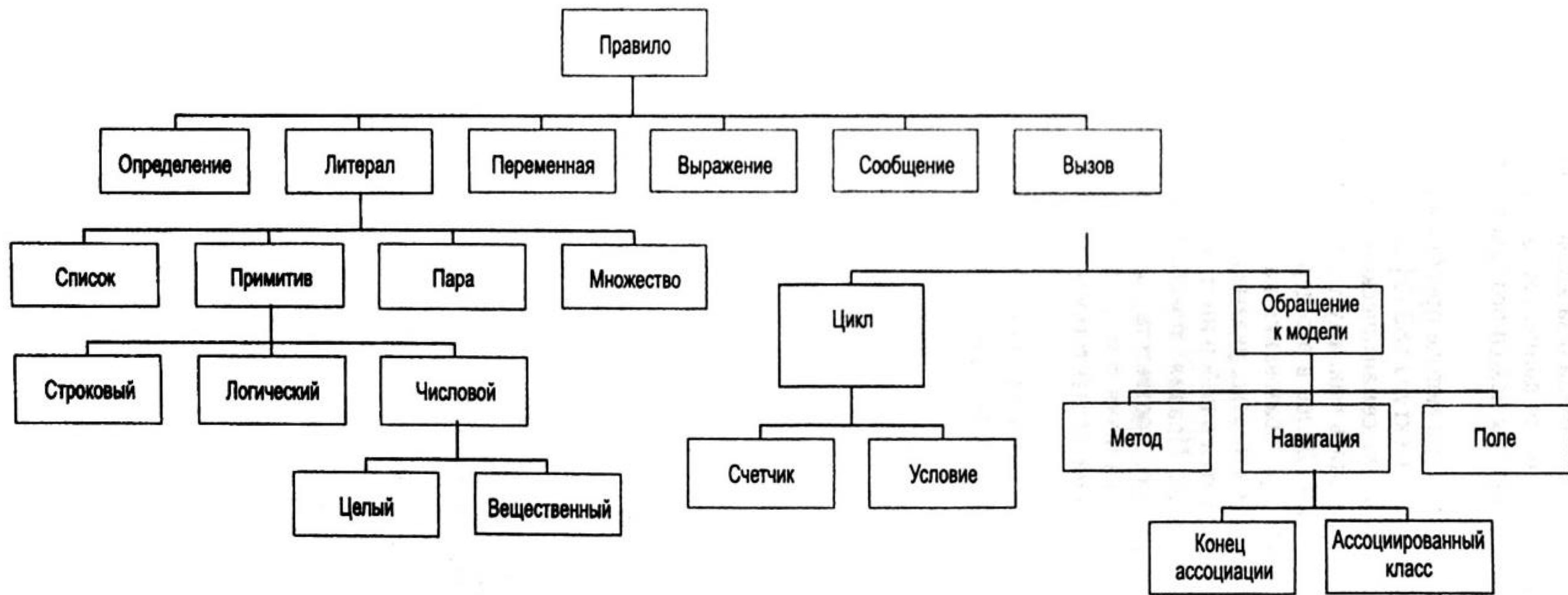


*Рис. 2.* Дерево семантического разбора с элементами вызова модели

Результатом стадии семантического анализа является синтаксическое дерево, дополненное элементами извлечения данных модели, а также таблица символов, содержащая все идентификаторы исходного текста программы с указанием типа для каждого.

Генератор целевого кода основан на методе преобразования дерева [8]. Целевой архитектурой кода выбрана архитектура x86 [9]. Данный генератор преобразует дерево, полученное после этапа семантического анализа, таким образом, чтобы каждый узел дерева мог работать максимум с двумя ячейками памяти или регистрами для хранения объектов. Далее в это дерево вставляются адреса времени выполнения. После такого преобразования каждый лист дерева содержит константу, регистр, ячейку памяти и индекс, указывающий значение атрибута.

Полученное дерево транслируется в набор инструкций целевой архитектуры, используя правила генерации кода. Правила преобразования представляют собой шаблон поддерева и инструкцию, соответствующую данному шаблону. Применяя правила ко всему дереву, реализуя тем самым свертку дерева, генератор получает готовый набор инструкций, соответствующий исходному тексту выражения языка fOCL.

Необходимо отметить, что при возникновении ошибки в тексте компилируемого правила на любом этапе компиляции разработчику генерируется соответствующее сообщение.

***Выводы.*** Создан язык fOCL, позволяющий описывать правила верификации проектов по разработке программного обеспечения в команде с использованием объектно-ориентированного моделирования.

Реализован компилятор языка fOCL, состоящий из четырех модулей. В настоящее время разработанный компилятор находится на этапе опытной эксплуатации.

СПИСОК ЛИТЕРАТУРЫ

1. Object Management Group (OMG) (2005, July 05). Unified Modeling Language infrastructure specification Version 2.0 [WWW document]. URL http://www.omg.org/spec/UML/2.0/Infrastructure/PDF/

2. Object Management Group (OMG) (2008, April 01). Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0 [WWW document]. URL http://www.omg.org/spec/SPEM/2.0/PDF

3. Object Management Group (OMG) (2006, January 01). Meta Object Facility (MOF) Core specification Version 2.0 [WWW document]. URL http://www.omg.org/spec/MOF/2.0/PDF/

4. Object Management Group (OMG) (2006, May 01). Object Constraint Language Specification (OCL) Version 2.0 [WWW document]. URL http://www.omg.org/cgi-bin/doc?formal/2006-05-01

5. Ахо А. В., Сети Р., Ульман Д. Д. Компиляторы: принципы, технологии и инструменты. М.: ИД Вильямс, 2003. 768 с.

6. Object Management Group (OMG) (2007, December 01). XML Metadata Interchange (XMI) specification Version 2.1.1 [WWW document]. URL http://www.omg.org/spec/XMI/2.1.1/PDF/

7. Фостер Дж. Автоматический синтаксический анализ. М.: Мир, 1975. 75 с.

8. Cattell, R. G. G. Automatic derivation of code generators from machine descriptors. TOPLAS 2:2, 1980. 173-190 с.

9. Поворознюк А.И. Архитектура компьютеров. Архитектура микропроцессорного ядра и системных устройств: Учеб. пособие. Ч.1. Харьков: Торнадо, 2004. 355 с.