УДК 007:681.3

ФОРМИРОВАНИЕ БАЗЫ МЕТАИНФОРМАЦИИ НА ОСНОВЕ СПЕЦИФИКАЦИЙ КАНОНИЧЕСКОЙ МОДЕЛИ

С.А. Ступников, Л.А. Калиниченко

ИПИ РАН, 117900, Москва, ул. Вавилова, 30/6
факс 9304505, телефон 3324933
E-mail: ssa@ipi.ac.ru, leonidk@synth.ipi.ac.ru

The component design is a process of refinement the requirement specifications by component specifications expressed by means of a canonical model. The model is used to uniform different information representation models. Metainformation base is a component specification repository, i.e. a primary resource for component design program tools. This paper considers the problems of forming the metainformation repository on the base of canonical model (SYNTHESIS language) text specifications.

Композиционное проектирование заключается в уточнении спецификаций требований спецификациями компонентов выраженных, средствами канонической модели. Каноническая модель используется для унификации различных моделей представления информации. База метаинформации есть репозиторий спецификаций компонентов, т.е. основной ресурс для средств композиционного проектирования. В статье рассматриваются проблемы формирования базы метаинформации на основе текстовых спецификаций канонической модели (языка СИНТЕЗ).

Введение

Одной из важнейших особенностей современных информационных систем (ИС) является тенденция перехода к конструированию ИС из готовых объектных компонентов, возможно, распределенных в сети [4,5,6]. Большое значение при этом приобретает понятие канонической модели представления данных. Каноническая модель призвана обеспечить формализованное представление требований и моделей анализа ИС, формализованное описание предметных областей, однородное (каноническое) представление разнородных компонентов, поддержку семантического согласования компонентов с целью их композиции.

Каноническая модель должна обладать формальной интерпретацией. Отображение канонических спецификаций в формальные позволяет строить формальные модели прикладных областей и компонентов, отражающие их статические и динамические свойства. Формальные спецификации и соответствующие инструментальные средства позволяют формализовать доказательство непротиворечивости спецификаций, а также доказательство корректности конкретизации спецификации требований композицией спецификаций компонентов.

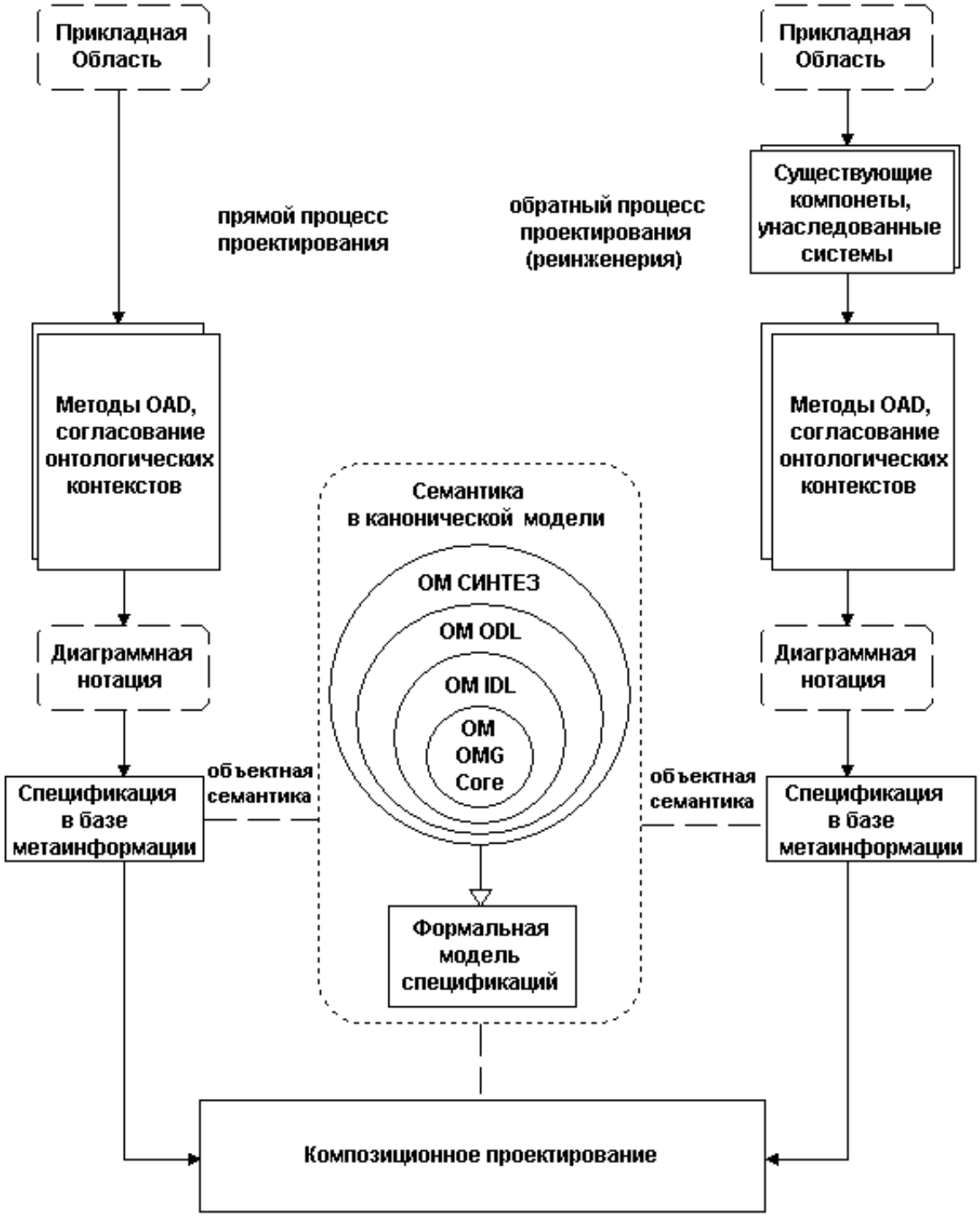
В данной статье[1] в качестве канонической модели рассматривается СИНТЕЗ [1] - язык спецификации и проектирования интероперабельных сред неоднородных информационных ресурсов.

Язык разработан в рамках проекта СИНТЕЗ [9]. В задачи проекта входит создание единообразного набора методов и средств композиционного проектирования ИС из существующих компонентов.

На рисунке 1 изображена логическая схема метода композиционного проектирования, разрабатываемого в рамках проекта СИНТЕЗ. На рисунке подчеркивается, что в процессе как прямого (на левой стороне рисунка), так и обратного проектирования (на правой стороне рисунка)следует принимать во внимание прикладные области, определяющие прикладнуюсемантику спецификаций требований или спецификаций компонентов.Компоненты могут интерпретироваться в контексте некоторого приложения, а значит, между соответствующими прикладными контекстами должно быть установлено соответствие, для чего в обоих случаях должны быть сформированы онтологические спецификации.

Для планирования требований, системного анализа и реинженерии используются традиционные средства объектного анализа и проектирования и диаграммные нотации. Результаты прямого и обратного проектирования служат входными данными для композиционного проектирования [4,5]. В качестве объектной семантики для диаграммной нотации (UML, расширенный средствами онтологических спецификаций и спецификаций операций) служат каноническая модель СИНТЕЗ и ее отображение в Нотацию Абстрактных Машин (Abstract Machine Notation, AMN) [2] - формальный язык спецификаций, основанный на типизированной теории множеств и логике первого порядка. B-технология [3] предоставляет методы и инструментальные средства для формального доказательства непротиворечивости спецификаций AMN и факта

[1] Работа выполнена в рамках проекта РФФИ, грант 01-07-90084



удовлетворения построенной спецификации системы требованиям. Более подробно процесс композиционного проектирования описан в [4].

Рисунок 1. Логическая схема композиционного проектирования

Основным ресурсом инструментальных средств автоматизации композиционного проектирования, хранящим метаинформацию о существующих компонентах, выраженную в виде формальных спецификаций, является база метаинформации. База хранит спецификации требований, спецификации существующих компонентов, промежуточные спецификации, возникающие в процессе проектирования. Инструментальные средства отображения канонической модели в AMN также используют базу как источник спецификаций.

База метаинформации для хранения формальных спецификаций языка СИНТЕЗ, средства ее поддержки и доступа к ней были разработаны в рамках проекта СИНТЕЗ. Средства композиционного проектирования, разработанные в проекте [4], используют базу как ресурс спецификаций. На рис. 1 показано, как попадают в

базу метаинформации спецификации требований (компонентов), определенные в диаграммной нотации (UML). Наряду с этим, важным является непосредственное формирование базы метаинформации на основе спецификаций, заданных в виде текстов в канонической модели. Именно этому способу помещения спецификаций в базу метаинформации посвящена настоящая статья.

Формирование базы метаинформации на основе спецификаций канонической модели есть задача специального загрузчика. Получая на вход такую спецификацию, загрузчик должен проверить ее корректность и сформировать в базе набор объектов, соответствующий данной текстовой спецификации. Данная статья написана на основе опыта построения такого загрузчика.

Родственной данной можно считать работу [7], в которой рассмотрена структура репозитория спецификаций компонентов как информационно-поисковой системы. Проблема поиска компонентов рассматривается узко - для компонентов-функций, представляемых отношениями, содержащими все допустимые пары входных/выходных значений функций.

Текст статьи организован следующим образом. В разделе (1) рассматриваются основные черты канонической модели и ее представления в базе метаинформации. В разделе (2) описываются средства и принципы построения формальной грамматики канонической модели и соответствующих грамматике абстрактных синтаксических деревьев. В разделе (3) рассматриваются методы формирования объектов базы метаинформации на основе таких синтаксических деревьев. В заключении подводится итог выполненной работы.

1. Каноническая модель и ее представление в базе метаинформации

Каноническая модель данных СИНТЕЗ - язык, ориентированный на семантическую интероперабельность и композиционное проектирование информационных систем в широком диапазоне существующих неоднородных информационных ресурсов. Этот язык обладает гибридными возможностями, обеспечивающими интеграцию как структурированных, так и слабоструктурированных моделей данных.

Язык включает средства представления фреймов - абстрактных значений специального вида, предназначенных для описания слабоструктурированной информации; все спецификации канонической модели имеют вид фреймов. Система типов включает универсальный конструктор произвольных абстрактных типов данных, а также представительный набор встроенных типов. Для типов определяется отношение специализации типа (подтип). Для представления множеств однородных сущностей в прикладных областях в языке используются классы. Экземпляры классов имеют специфический тип. Для описания параллельного и асинхронного поведения прикладных систем и сред интероперабельных ресурсов как систем динамических дискретных событий в языке используются средства представления потоков работ.

Язык включает также средства представления логических формул многосортного объектного исчисления (типизированного языка первого порядка), использующиеся для запросов к интегрированным наборам цифровых коллекций, а также для задания ограничений целостности и поведения.

Ниже приводится пример спецификации абстрактного типа данных (АТД) *Society*.

```
{ Society; in: type;
  supertype: Organization;
  name: string;
  usoc:
   {in: predicate, invariant;
    {{all s1/Society, s2/Society(s1.name = s2.name => same(s1,s2)}}
   };
  in country: string;
  staff: {set; type of element: Person;};
}
```

В соответствии с данной спецификацией, АТД *Society* имеет строковые атрибуты *name* и *in_country*, атрибут *of_proposal*, атрибут *staff* типа множества элементов типа *Person*, а также инвариант *usoc*, утверждающий уникальность имени общества.

Для хранения спецификаций канонической модели используется база метаинформации, реализованная в проекте СИНТЕЗ на основе СУБД Oracle 8i. В базе хранятся метаобъекты, соответствующие канонической модели. Структура базы представляет собой описание метамодели для разных видов метаобъектов. Средства описания метаданных включают модель фреймов, являющуюся унифицированной основой для представления произвольных видов информации, объектную модель с поведенческими спецификациями, описания классов объектов. На рисунке 2 изображены отношения между фреймами, типами и классами как сущностями метамодели.

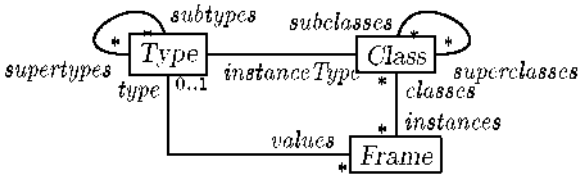


Рисунок 2. Фреймы, типы и классы

В качестве примера представления сущностей канонической модели в базе метаинформации на рисунке 3 изображена частичная UML-диаграмма типа *ADT*, метаобъекта, представляющего абстрактный тип данных канонической модели.

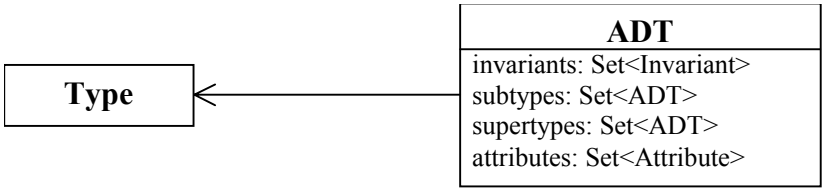


Рисунок 3. Тип ADT

Спецификация АТД является спецификацией типа, а потому тип *ADT* является прямым подтипом типа *Type*. Спецификация АТД может содержать инварианты, супертипы и описания атрибутов АТД; данный АТД также может быть супертипом для других АТД. Поэтому тип *ADT* содержит соответствующие атрибуты для хранения инвариантов, супертипов, подтипов и атрибутов.

Для обращения к базе предусмотрена библиотека с Java-интерфейсом, обеспечивающим создание, изменение, удаление метаобъектов, их поиск и работу с множествами метаобъектов. Для приведенного в качестве примера типа *ADT* предусмотрен интерфейс *ADTDef*. Одним из методов данного интерфейса является, например, *get_subtypes()*, использующийся для доступа к подтипам данного абстрактного типа. Метод возвращает множество(*java.util.Set*) подтипов данного типа. Для создания нового метаобъекта типа *ADT* используется метод *new_ADT()* интерфейса *DB*. Интерфейс *DB* является средством доступа к базе информации в целом.

2. Формальная грамматика канонической модели

Первой из задач загрузчика, осуществляющего формирование базы метаинформации на основе спецификаций канонической модели, является проверка синтаксической корректности спецификаций - проверка соответствия грамматике языка. Для решения этой задачи используются так называемые генераторы парсеров - программные средства, на основании формальной грамматики автоматически строящие объектный код парсера (т.е. лексико-синтаксического анализатора) для данной грамматики. Для генерации парсера канонической модели используется программный продукт *antlr 2.7.1*[8], порождающий объектный код парсера на языке *Java*.

Таким образом, возникает задача построения формальной грамматики языка канонической модели, являющейся корректными входными данными для *antlr*. При решении этой задачи за основу грамматики было взято описание языка [1], которое затем было подвергнуто более строгой формализации.

Начальным этапом формализации является выделение в языке набора лексем (неделимых единиц языка). Поток лексем затем подается на вход синтаксическому анализатору. Пример определения лексем приведен ниже.

```
DIGIT : '0'..'9' ; LETTER: 'a'..'z' |  'A'..'Z';
IDENTIFIER : LETTER (( UNDERSCORE )? ( LETTER | DIGIT ))*;
COLON : ':'  ; SEMI : ';'  ; COMMA : ','  ; LBRACE : '{'  ; RBRACE : '}';
```

При определении лексем цифры, буквы и идентификатора использованы оператор выбора альтернатив '|' и оператор опциональности '?'. Определены также лексемы, соотвествующие разделительным символам языка.

Следующим этапом формализации является выделение набора синтаксических правил грамматики на основании описания языка и средств, предлагаемых генератором парсеров. Такие синтаксические правила являются основой синтаксического анализатора. Нетерминальные символы описания языка являются возможными кандидатами имен синтаксических правил. Основной задачей данного этапа является устранение недетерминизма в синтаксических правилах. В качестве примера можно привести следующее правило языка

```
class_name : class_identifier | metaclass_identifier | ...
```

когда нетерминалы, использованные как альтернативы в операторе выбора, имеют одинаковый синтаксис:

```
class_identifier : IDENTIFIER;
metaclass_identifier : IDENTIFIER;
```

В этом случае парсер не сможет самостоятельно выяснить, какое правило следует применить к лексеме *IDENTIFIER*, поданной на вход. Недетерминизм в данном случае может быть разрешен на основе семантики анализируемых конструкций - именем класса может быть идентификатор класса либо идентификатор метакласса, хотя синтаксически они оба представляют собой идентификатор.

Вообще говоря, привнесение семантики является основной причиной возникновения недетерминизма в синтаксических правилах. Обобщением вышеприведенного примера недетерминизма является следующее правило

```
rule : subrule_1 | ... | subrule_n;
```

где множества альтернатив правил subrule_1 ... subrule_n имеют непустое пересечение: когда на вход парсеру подаются лексемы из этого пересечения, выбор между правилами неоднозначен. Одним из выходов является установление в парсере опции более глубокого анализа входной последовательности лексем. При глубине анализа *n* при каждом возникновении необходимости выбора правила синтаксический анализатор рассматривает *n* необработанных лексем из потока, и на основании того, какие лексемы составляют последовательность, выбирает правило из набора альтернатив. В ситуации, когда нужная глубина анализа неизвестна, полезной является конструкция синтаксического предиката

```
rule :
( prediction block\ 1 ) => subrule 1 |
( prediction block\ 2 ) => subrule 2
;
```

Здесь парсер проверяет входную последовательность лексем на соответствие синтаксическим правилам *prediction_block_1* и *prediction_block_2*; в зависимости от того, какому из этих правил соотвествует входная последовательность, выбирается правило *subrule_1* или *subrule_2*.

В случае недостаточности для элиминации детерминизма вышеприведенных средств, применяется коррекция грамматики. Части правил, участвующих как альтернативы в операторе выбора, имеющие одинаковую синтаксическую структуру, могут быть объединены в одно правило и выброшены из исходных правил. В этом случае семантические различия в правилах, из-за которых возник недетерминизм, должны быть отслежены при дальнейшем семантическом анализе загружаемой спецификации. Необходимая коррекция грамматики также может быть произведена с помощью элиминации некоторых нетерминалов и выноса их синтаксической структуры на внешние уровни.

Так, правило *term* описания языка

```
term : {{ expression }} | actual_type | typed_variable | atom | entire_value | multiterm ;
```

после формализации приобрело вид

```
term :
 (  ( LBRACE LBRACE ) => LBRACE LBRACE expression RBRACE RBRACE
 ...
 |  (IDENTIFIER DOT "nonobject") => IDENTIFIER DOT "nonobject"
 |  ( compound identifier LPAREN ) => parameterized atom
 |  ( entire value ) => entire value
 );
```

В данном случае при формализации все альтернативы получили синтаксические предикаты, альтернативы *typed_variable* и *multiterm* были поглощены *entire_value*, *atom* расщепился на *parameterize_atom* и остаток, который был поглощен другими альтернативами, внутренняя структура *actual_type* частично вынесена на внешний уровень (альтернатива *IDENTIFIER DOT "nonobject"*).

Как было сказано выше, парсер осуществляет синтаксический анализ входного отока символов, проверку его соответствия формальной грамматике. Однако, для того, чтобы сформировать в базе метаинформации корректное множество метаобъектов, соответствующее текстовой спецификации канонической модели, необходим семантический анализ спецификации. Такой анализ нужен, в частности, из-за того, что семантика описания языка частично удаляется из грамматики в процессе формализации.

Семантический анализ синтаксически корректной спецификации проводится на основе временных структур данных, хранящих спецификацию. В качестве таких структур *antlr* предлагает двумерные абстрактные синтаксические деревья (АСД). В формальную грамматику может быть добавлен код, на основании которого парсер автоматически построит АСД входной спецификации. Построено может быть любое дерево, реализующее интерфейс *AST*(название интерфейса происходит от сокращения Abstract Syntax Tree, в русском переводе - абстрактное синтаксическое дерево, АСД). Ниже указаны некоторые из методов интерфейса.

```
public interface AST {
// метод возвращает тип данной вершины дерева
public int getType();
// метод возвращает текст данной вершины дерева
public String getText();
// метод возвращает первого потомка данной вершины дерева
public AST getFirstChild();
// метод возвращает брата, следующего за данной вершиной дерева
public AST getNextSibling();
```

```
      ...\\
      \}
```

Как видно из спецификации интерфейса, дерево имеет типизированные вершины, в которых может храниться некоторый текст. Имеются также методы, осуществляющие навигацию по дереву.

Вероятными кандидатами на вершины дерева являются нетерминальные символы, т.е. имена правил грамматики. Для повышения эффективности следует по возможности минимизировать число вершин без ущерба для семантики спецификации. Рекурсивные структуры также следует по возможности заменять на нерекурсивные. Примером является правило *metaclass_name_list* в описании языка, имеющее вид

```
metaclass_name_list : metaclass_name ( , metaclass_name_list )*;
```

Оператор '*' здесь означает произвольное количество повторений конструкции. В формальной грамматике рекурсивная структура правила заменена на однородную, а правило *metaclass_name* не получило отдельной вершины (такая вершина не добавляет семантической информации).

```
metaclass name list :
 metaclass_name ( COMMA metaclass_name )*
{ \#metaclass_name_list = #([METACLASS, "Metaclass"], #metaclass_name_list); };
metaclass\_name : IDENTIFIER |  "metaclass" | "association" | "metatype" | "class";
```

Правило *metaclass_name_list* здесь посредством кода, заключенного в фигурные скобки, приобретает вершину типа *METACLASS*, потомками которой являются имена метаклассов. Подобным же образом формируется древовидная структура всей грамматики. Обсуждение семантического анализа получаемых АСД и формирования метаобъектов на основе АСД проводится в следующем разделе статьи.

Формирование базы метаинформации на основе абстрактных синтаксических деревьев

На основе формальной грамматики *antlr* генерирует Java-классы лексического и синтаксического анализаторов, в случае грамматики канонической модели эти классы названы соответственно *SynthesisLexer* и *SynthesisParser*. Класс загрузчика *SynthesisLoader* как контейнер включает в себя классы *SynthesisLexer* и *SynthesisParser* и управляет их работой. *SynthesisLoader* подает файл текстовой спецификации канонической модели на вход *SynthesisLexer* и получает АСД спецификации при помощи метода *SynthesisLoader.getAST()*.

Далее *SynthesisLoader* передает управление классам, задача которых - обойти АСД и с помощью прикладного программного интерфейса базы метаинформации создать соответствующие спецификации объекты базы, произведя при этом проверку внутренней непротиворечивости спецификации и непротиворечивости спецификации относительно базы метаинформации: одни части спецификации могут ссылаться на другие части спецификации, а также на метаобъекты, уже присутствующие в базе. Непротиворечивость спецификации заключается в корректности таких ссылок, т.е. в существовании соответствующих объектов базы или частей спецификации. В случае некорректности ссылок классами документируется сообщение о соответствующей ошибке.

Принципы структуризации кода классов обхода АСД заключаются в следующем. Для каждого типа вершин АСД (т.е. практически, для большинства синтаксических правил) создается соответствующий метод в каком-либо классе обхода. Объединение методов в классы производится на основе анализа логической общности, зависимости методов друг от друга. В случае, когда некоторое синтаксическое правило используется при определении нескольких других различных правил, его следует выделить в отдельный класс. В отдельный класс также выделяются утилиты поиска такие как поиск типа в модуле или атрибута в типе.

Примером является структуризация методов класса *AbstractTypeClass*. Часть грамматики, соответствующая абстрактному типу данных канонической модели, выглядит следующим образом:

```
abstract type :
LBRACE type_identifier SEMI "in" COLON "type" ... SEMI
( "params" COLON LBRACE formal_parameter_list RBRACE SEMI )?
( "supertype" COLON supertype list SEMI )?
( "rename" COLON LBRACE adt renaming list RBRACE SEMI)?
( attribute specification list )? RBRACE
{ #abstract_type = #([ABSTRACTTYPE, "AbstractType"], #abstract_type); };
```

Методы *abstractType*, *adtRenamingList*, *adtRenamingListElement*, соответствующие правилам *abstract_type*, *adt_renaming_list*, *adt_renaming_list_element*, объединены в класс *AbstractTypeClass*. Методы, соответствующие правилам *formal_parameter_list*, *supertype_list*, *attribute_specification_list*, вынесены в отдельные классы, поскольку используются при определении спецификаций фреймов, функций, классов, процессов канонической модели.

Структура метода, соответствующего типу вершины АСД (синтаксическому правилу), прослеживается на примере метода *abstractType*.

```
public ADTDef abstractType(CommonAST ast){
```

```
CommonAST cursor = (CommonAST)ast.getFirstChild();
ADTDef adt = null;
try{
 for ( ; cursor != null; cursor = (CommonAST)cursor.getNextSibling()){
  switch (cursor.getType()){
   case TYPEIDENTIFIER :
    ... adt = (ADTDef)TypeIdentifierClass.typeIdentifier(cursor);
    if ( adt == null ){
     adt = SynthesisASTLoader.db.new adt(); adt.set name(cursor.getFirstChild().getText()); ...
    } ...
    break;
   case PARAMS :
    ... FormalParameterClass.params( cursor, adt.get parameters() ); ... break;
   case SUPERTYPELIST :
    ... SupertypeListClass.supertypeList( cursor, adt.get supertypes() ); ... break;
   case ADTRENAMINGLIST :
    ... adtRenamingList( cursor, adt ); ... break;
   case ATTRIBUTESPECIFICATIONLIST :
    ... AttributeSpecificationListClass.attributeSpecificationList( cursor, adt ); ... break;
  }
 }
} catch (SQLException ex) { ... }
return adt;
}
```

Метод получает в качестве входного параметра вершину дерева, просматривает ее вершины-потомки, и в зависимости от их типа передает управление другим методам. Задача *abstractType* - сформировать в базе метаинформации объект типа *ADT*, соответствующий АТД, определенному спецификацией в случае ее непротиворечивости. Метод решает эту задачу, используя интерфейс базы и вызываемые методы обхода дерева. В примере для краткости оставлен лишь код, создающий и именующий метаобъект типа *ADT* в случае, если метод *typeIdentifier* не находит в базе соответствующего загруженного типа.

Как было замечено в разделе 2, при формализации описания языка и элиминации недетерминизма, из грамматики исчезают некоторые семантические особенности. Такие случаи необходимо отслеживать при семантическом анализе АСД. Примером является метод *typeDesignator*. Правило описания языка, соответствующее этому методу, выглядит следующим образом:

```
type designator : type name | attribute name | ... ;
type name : class identifier | ... ;
class_identifier : module_identifier DOT IDENTIFIER;
attribute_name : attribute_type_identifier DOT IDENTIFIER;
```

Здесь указатель типа семантически может быть именем типа (а значит, идентификатором класса) либо именем атрибута (что означает ссылку на тип этого атрибута). Синтаксически, однако, такие конструкции неразличимы, и в формальной грамматике выглядят как

```
IDENTIFIER DOT IDENTIFIER
```

Поэтому метод обхода последовательно перебирает возможные семантические ситуации. Исходя из альтернативы *type_name* с помощью утилит поиска ищется класс в соответствующем модуле. В случае неудачи согласно альтернативе *attribute_name* ищется тип, если тип найден, проверяется, является ли он *ADT*, и в типе производится поиск атрибута. Если соответствующий атрибут найден, то его тип возвращается методом. Ниже приведена часть кода, соответствующая второй альтернативе.

```
...
type def = SearchUtilitesClass.searchType(cursor.getText(), ...);
if ( type_def == null ) { ... // Error: type not found }
else{
 if ( type def.is adt() != true ) { ... // Error: type is not ADT }
 else{
  AttributeDef attribute =
   SearchUtilitesClass.searchAttribute( cursor.getNextSibling().getText(), (ADTDef)type def );
  if ( attribute != null ){ return attribute.get_type(); }
  else\{ ... // Error: attribute not found }
 }
}
```

Похожим образом обрабатываются и более сложные семантические ситуации.

Заключение

В работе описаны принципы и средства формирования базы метаинформации, предназначенной для хранения спецификаций информационных ресурсов Формирование базы осуществляется на основе текстовых

спецификаций канонической модели СИНТЕЗ. Каноническая модель призвана унифицировать спецификации ресурсов и обеспечивать их полноту в целях композиционного проектирования.

Рассматриваются принципы формализации описания языка канонической модели, построения формальной грамматики, пригодной для генерации лексико-синтаксического анализатора (парсера) с помощью программного средства *antlr* а также средства построения абстрактных синтаксических деревьев на основе грамматики. Такие деревья рассматриваются как временные структуры данных, используемые при семантическом анализе спецификации.

Определены принципы формирования набора объектов базы метаинформации на основе абстрактных синтаксических деревьев, соответствующих текстовой спецификации канонической модели.

Литература

1. *Калиниченко Л. А.* СИНТЕЗ: Язык определения, проектирования и программирования интероперабельных сред неоднородных информационных ресурсов. - М.: ИПИ РАН, 1993б, 113 с.
2. *Abrial J.-R.* The B-Book: assigning programs to meaning. - Cambridge University Press, 1996, 773 с.
3. *Abrial J.-R.* B-Technology. Technical overwiew. - BP International Ltd., 1992, 73 p.
4. *Briukhov D.O., Kalinichenko L.A.* Component-based information systems development tool supporting the SYNTHESIS design method. - Proceedings of the East European Symposium on "Advances in Databases and Information Systems" (ADBIS'98), September 1998, Poland, Springer, LNCS N 1475, 1998, p. 305-327
5. *Kalinichenko L. A.* Compositional Specification Calculus for Information Systems Development. - Proceedings of the East-West European Symposium on Advances in Databases and Information Systems (ADBIS'99), Maribor, Slovenia, September 1999, Springer Verlag, LNCS, p. 317-331
6. *Jacobson I., Griss M., Jonsson P.* Software Reuse. - ACM Press , 1997, 497 p.
7. *Mili R., Mili A., Mittermeir R. T.* Storing and Retrieving Software Components: A Refinement Based System. - IEEE Transactions on Software Engineering, vol. 23, no. 7, July 1997. p. 445-460
8. Генератор синтаксических анализаторов *antlr.* - http://www.antlr.org
9. Проект СИНТЕЗ. - http://www.ipi.ac.ru/synthesis/projects/SYNTHESIS