АВТОМАТИЗАЦИЯ СИНТЕЗА ЯДРА ИНФОРМАЦИОННОЙ СИСТЕМЫ С ИСПОЛЬЗОВАНИЕМ UML-ОПИСАНИЯ*

Е. А. Черкашин, Р. К. Федоров, И. В. Бычков, В. В. Парамонов
Институт динамики систем и теории управления СО РАН, Иркутск, Россия
e-mail: {eugeneai,fedorov,bychkov,slv}@icc.ru

An approach to automation of the syntheses of a kernel of the information system with application to an organization utilizing OMG MDA architecture is considered. The synthesis means a code generation of the subsystem modules of the information system according to the prescribed UML model. The synthesis software is implemented as an artificial intelligence system. The generated code results from analysis of the UML-diagrams of the system's knowledge base.

Введение

Современные информационные системы (ИС) ранга предприятия строятся, как правило, по общей схеме, где ИС состоит из трех основных подсистем. *Хранилище информации* обеспечивает хранение данных ИС в определенном формате и доступ к этим данным. *Уровень управления приложением*, называемый иногда "уровнем бизнес-логики", представляет собой модель взаимодействия объектов предметной области, реализованную в виде программы. Он, в частности, реализует корректное с точки зрения предметной области изменение информации в хранилище, а также ее проверку на непротиворечивость. *Интерфейс пользователя* отображает информацию в ИС в удобном для пользователя виде, передает события, инициируемые пользователем, уровню управления приложением. Большинство ИС включают также подсистему генерирования производных отчетов.

В настоящее время выделяется несколько подходов, позволяющих создавать сложные ИС, использование которых может значительно улучшить качество, сократить стоимость и время поставки ИС, в том числе это:

— компонентная технология разработки моделей ИС:

— визуальное представление различных аспектов проекта, например визуальное моделирование с использованием CASE (Computer Aided Software Engineering)-средств.

Примерами первого подхода выступают известные системы SAP/DB, R/3, PL-SQL ORACLE. Библиотеки программных модулей уже включают реализации подсистем, профессионально выполненных на достаточно высоком абстрактном уровне. В задачи разработ-

*Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (гранты № 04-07-90227-в, № 05-07-97201-в, № 05-07-97204-в).

чика ИС входят комбинирование и дальнейшая специализация модулей библиотеки на своем предприятии. Будем считать, что системы RAD (Rapid Application Development) относятся к системам этого типа. Известные представители RAD — это средства разработки приложений Borland Delphi и Borland C++ Builder, Microsoft Visual Development Studio.

Технология визуального моделирования, в том числе CASE, позволяет работать со сложными системами и проектами, представляя ИС в виде некоторой формализованной модели. В основе CASE-подхода лежит идея генерирования части кода программы по такому формальному описанию. В процессе проектирования ИС, базирующихся на объектно-ориентированном представлении, используется распространенный стандарт представления — унифицированный язык моделирования (Unified Modeling Language, UML) [1].

Встроенные в CASE-системы генераторы программного кода переводят элементы UML-диаграммы, представляющие объекты ИС, в программный код. Процедуры порождения программного кода, как правило, являются независимыми, т. е., например, SQL-скрипты создаются отдельно от объектной структуры уровня приложения. Чаще всего создаются две диаграммы — одна для объектной структуры, другая — для базы данных. Авторам не приходилось встречать упоминание о генерировании какого-либо пользовательского интерфейса по диаграммам UML. Обычным методом решения этой задачи является анализ структуры базы данных.

Визуальное моделирование позволяет преодолеть проблему сложности программного обеспечения. И не важно, преобладает в проекте “техническая сложность” (статическая) или “динамическая сложность управления”. Сложность программных систем возрастает по мере создания новых версий. И в какой-то момент наступает “эффект критической массы”, когда дальнейшее развитие ИС становится невозможным, поскольку уже никто не представляет в целом “что и почему происходит”. Визуальные модели обеспечивают ясность представления выбранных архитектурных решений на удобном для восприятия человека уровне, что позволяет воспринимать разрабатываемую систему в комплексе, содержательно организовать общение между заказчиками и разработчиками. При этом становится возможным достижение таких целей, как повышение качества программного продукта, сокращение стоимости проекта, поставка системы в запланированные сроки.

Дальнейшим развитием идеи визуального проектирования выступает новый подход к проектированию программных систем, называемый MDA (Model Driven Architecture) [2] и предложенный в 2001 году консорциумом OMG (Object Management Group) [3]. MDA преследует целью отделить спецификацию функций системы от способа реализации этих функций в различных платформах. MDA призвано обеспечить подход к созданию программных систем через:

— спецификацию системы независимо от платформы, на которой она будет функционировать;

— спецификацию собственно платформы;

— выбор конкретной платформы для системы;

— преобразование спецификации системы в конкретную систему, функционирующую на конкретной платформе.

Три основные цели, преследуемые MDA, — это переносимость, открытость к взаимодействию с другими системами и повторное использование.

Поясним идею на простом примере. В UML-диаграмме ИС возможно задать тот факт, что один из классов является справочником и, соответственно, на другой объект можно ссылаться через один из его ключевых атрибутов (ассоциация “многие-к-одному”), что,

в свою очередь, является стандартным способом связывания таких реляционных таблиц в реляционных базах данных. Стандартным способом интерпретации представленной ассоциации в CASE-системах является преобразование ее в контексте SQL-баз данных в первой таблице в поле-ссылку на первичный ключ другой таблицы. В MDA, кроме этого, учитываются дополнительные свойства, например, в виде создания пользовательского интерфейса, позволяющего выбрать значение поля из набора возможных значений, хранящихся в справочнике.

В основе подхода лежит понятие *модели, не зависящей от платформы* (Platform Independent Model — PIM). PIM описывает систему на том уровне абстракции, которая позволяет рассматривать ее как не зависящую от конкретных свойств платформы. Примерами PIM выступают программные системы, выполняемые на виртуальных машинах (например, приложения Java). Одним из способов представления PIM является UML. Для того чтобы PIM могла быть реализована, необходимо преобразовать ее в модель, учитывающую особенности платформы. Эта модель, в свою очередь, называется *моделью, зависящей от платформы* (Platform Specific Model — PSM). В последнем примере указанное преобразование осуществляет виртуальная машина. Преобразование PIM в PSM осуществляется при помощи формального представления свойств платформы, *модели платформы* (Platform Model). В соответствии с MDA-подходом PSM является описанием ИС, представляющим ее на уровне, более абстрактном, чем непосредственно программный код.

В данной работе предлагается методика преобразования PIM в PSM и далее в программный код, причем:

— представление PIM осуществляется при помощи UML;

— модель платформы представляется при помощи базы знаний;

— преобразование PIM в PSM осуществляется поэтапно с использованием процедур логического вывода и шаблонов программного кода.

1. Представление PIM

Для представления PIM используются диаграммы UML. В частности, диаграмма классов представляет программные объекты и связи (ассоциации) между этими объектами. В соответствии со стандартом UML определение каждого объекта в диаграмме классов может включать заголовок, список атрибутов и список методов. Между объектами задаются ассоциации, при помощи которых моделируется межобъектное взаимодействие в общем виде. Объекты объединяются в классы, задаваемые комбинациями *стереотипов*.

Особенности предметной области ИС представляются через уточнение семантики объектов и ассоциаций PIM. В стандарте UML предусмотрен способ внесения в диаграмму дополнительной информации. Для этого используются стереотипы и *теговые значения*. Стереотипы предназначены прежде всего для группировки классов объектов (метаклассификации), например, при помощи стандартного стереотипа “`interface`” задается классификатор [1], называемый интерфейсом. Теговые значения позволяют задавать именованные значения для отдельных элементов диаграмм. Например, для атрибута `name` при помощи тегового значения `caption` задается название “Наименование” этого атрибута в пользовательском интерфейсе, а при помощи стандартного тегового значения `documentation` для этого атрибута задается текст-комментарий “Стандартное название лекарства”. Стандарт UML позволяет задавать стереотипы и теговые значения для всех элементов диаграммы.

1.1. Ограничения

Ограничения свойств объектов ИС являются важной частью PIM. Примерами реализации ограничений выступают типы данных, диапазоны значений, шаблоны ввода данных (маски ввода), триггеры таблиц и т. д.

Имеются два способа определения ограничений: в подмножестве естественного языка и в некотором формальном языке задания ограничений, например OCL (Object Constraint Language) — языке ограничений над объектами [4]. Основной задачей, которую решает OCL, является определение ограничений на сущности и структуры, представленные в UML-диаграмме классов. OCL может применяться для определения инвариантных ограничений на значения полей класса, возвращаемых значений операций классов, связей между локальными переменными. Существует возможность описывать пре- и постусловия (`pre-` и `post`-conditions), в частности, для контроля входных и выходных параметров операций. Ограничения представляют собой логические условия, которые должны быть истинными в заданное видом ограничения время. OCL позволяет использовать в выражениях значение объекта до его изменения, так называемое `pre@`-значение.

При проектировании реляционных баз данных возможность определения пред- и постусловий операций и `pre@`-значений является существенным фактором. Анализ таких ограничений позволяет сгенерировать триггеры и встроенные процедуры, которые, в частности, должны отслеживать целостность базы данных при ее модификации. На основе инвариантных ограничений, например, создаются SQL-конструкции типа “CONSTRAINT”.

Под инвариантом класса в OCL понимается логическое выражение, вычисление которого должно давать true при создании каждого объекта данного класса и сохранять это значение в течение всего времени существования объекта. При определении инварианта требуется указать имя класса и выражение, определяющее инвариант указанного класса. Синтаксически это выглядит следующим образом:

```
context <class_name> inv:
<OCL выражение>
```

Здесь `<class-name>` — имя класса, для которого определяется инвариант, `inv` — ключевое слово, говорящее о том, что определяется именно инвариант, а не другой вид ограничений, и `context` — ключевое слово, которое говорит о том, что контекстом OCL-выражения являются объекты класса `<class-name>`, т. е. OCL-выражение должно принимать истинное значение для всех объектов этого класса. Отметим, что OCL — типизированный язык. Существуют трансляторы с этого языка в методы объектов или assert-условия.

Рассмотрим пример преобразования ограничения в программный код. Ограничение, заданное на естественном языке как “дата выписки из стационара должна быть не раньше даты поступления в стационар”, представляется в OCL следующим образом:

```
context Person inv:
self.out_date >= self.in_date
```

В SQL-базе данных преобразуется в оператор `ALTER TABLE PERSON ADD CONSTRAINT OUT_DATE>=IN_DATE;` в пользовательском интерфейсе в части кода проверки корректности входных данных синтезируется код: `if (! this->out_date>=this->in_date) { ... message (''неверная дата выписки'');}` Информационное сообщение ассоциируется с ограничением при помощи теговых значений.

1.2. Представление модели платформы

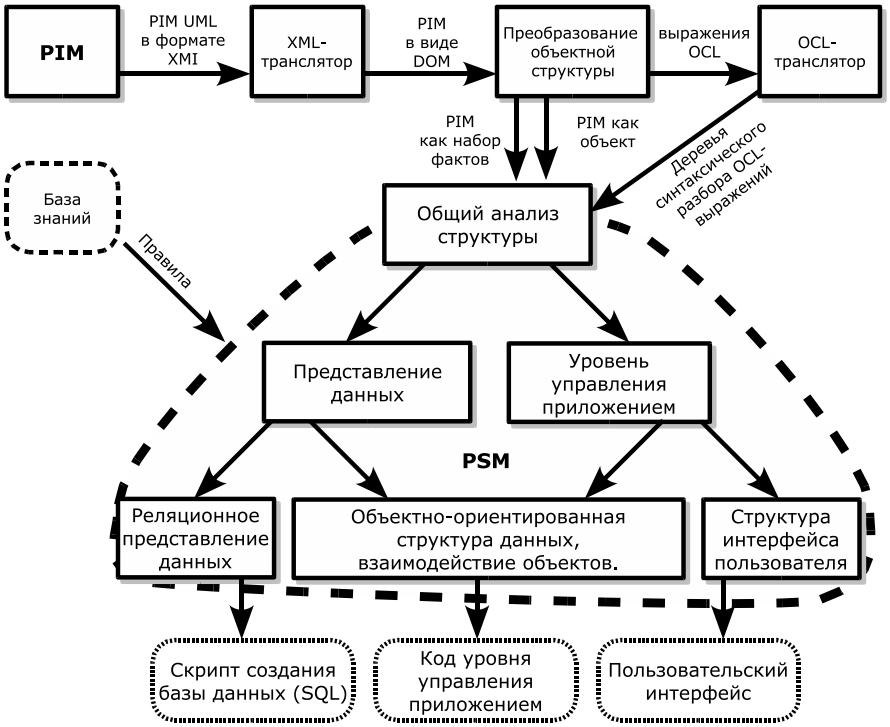
Преобразование PIM в PSM и генерирование исходного кода реализуются при помощи базы знаний системы и шаблонов программного кода. В базе знаний формализуются отношения между элементами диаграммы классов в контексте модели платформы и дополнительной информации, задаваемой стереотипами и теговыми значениями. Например, для класса, помеченного стереотипом "`abstract`" (класс, который не может иметь экземпляров), при создании SQL-скрипта базы данных не имеет смысла создавать специальную таблицу. Однако атрибуты этого класса наследуются потомками, т. е. попадают в соответствующие таблицы в виде атрибутов. Генерируется производный SQL-запрос (SQL View), выбирающий все экземпляры всех потомков абстрактного класса. Этот запрос становится методом соответстующего абстрактного класса на уровне приложений, т. е. по формализованному на UML-диаграмме абстракному классу порождается абстрактный класс на уровне приложений, но соответствующей таблицы в базах данных не создается. Таким образом, семантика стереотипа "`abstract`" (представление абстрактного класса в PSM) в контексте представления данных в виде SQL-таблиц соответствует SQL-запросу (программному коду), реализованному как View, а на уровне приложений — программному коду, определяющему этот абстрактный класс.

В приведенном примере принадлежность атрибута той или иной таблице определяется правилом, которое можно проинтерпретировать так: "Таблице принадлежит атрибут, если он принадлежит соответствующему неабстрактному классу UML-диаграммы либо одному из его непосредственных абстрактных классов-предков (родителей, родителей родителей и т. д.)". Реализация правила представлена в разд. 3

2. Архитектура системы анализа модели PIM и генерирования PSM

На рисунке представлены архитектура и схема взаимодействия функциональных модулей программы-генератора ядра ИС. UML-диаграмма создается при помощи какого-либо инструмента визуального редактирования и передается в систему в виде XMI-файла (Metadata Interface). Использование спецификации XMI позволяет упростить совместное использование метаданных различными инструментальными средствами и хранилищами. XMI объединяет в себе язык расширенной разметки Extensible Markup Language (XML) [5] с репозиторием метаобъектов Meta Object Facility (MOF) и универсальным языком моделирования UML. Метаданные можно описывать на UML, сохранять в MOF, а различные инструментальные средства и хранилища будут обмениваться ими посредством XMI. Транслятор XMI-файла — это стандартный траслятор, поддерживающий спецификацию DOM (Document Object Model). Полученная модель DOM преобразуется в две параллельные структуры данных: древо объектов языка императивного программирования Python [6] и набор связанных атомарных высказываний логического языка программирования Prolog. Выражения языка ограничений OCL экстрагируются из DOM и транслируются в деревья синтаксического разбора.

PIM поступает на вход иерархии связанных модулей анализа и генерирования программного кода, в корневой вершине располагается модуль анализа, осуществляющий распознавание базовых структур и свойств UML-диаграммы. Задача этого модуля — разобраться в структуре UML-диаграммы, проанализировать иерархию классов и типов дан-



Архитектура и схема взаимодействия модулей программы-генератора.

ных, выявить ассоциации, их свойства и т. д. На этом этапе, например, распознаются абстрактные классы, формируются списки предков и потомков класса. Результатом анализа в корневом модуле являются новые производные факты о структуре UML-диаграммы. Полученные производные факты представляют собой часть синтезируемой PSM: ее структуру, свойства элементов этой структуры. PSM передается на нижние уровни иерархии, где осуществляются дальнейшая ее специализация и непосредственно порождение программного кода. Над деревьями синтаксического разбора производятся преобразования в контексте создаваемой PSM. Например, ограничения, заданные в абстрактных классах, распространяются на классы-потомки. Далее осуществляется генерирование программного кода, реализующего ограничение.

3. Реализация модулей анализа и генерирования программного кода

Рассмотрим пример реализации одного из модулей анализа PIM и генерирования программного кода. В этом примере осуществляется создание SQL-скрипта таблицы базы данных для неабстрактных классов, при этом для каждого класса создается таблица базы данных, атрибуты таблиц соответствуют атрибутам классов.

Модуль реализован в виде класса языка Python-2.3. Язык программирования Python обладает средствами анализа объектной структуры программы во время исполнения. Программе доступна практически любая метаинформация, в том числе оформленная специальным образом текстовая документация о методах, классах и функциях. Правила языка Prolog встроены в методы этого класса в виде таких текстов. Запросы к правилам Prolog выполняются при помощи метода `query`. Этот метод является итератором, т. е. генериру-

ет наборы ответов от запрашиваемых правил. Перехватом ответов при помощи оператора цикла `for` реализуется генерирование соответствующего программного кода.

```
class RulesMixing:
    #Правила модуля генерирования SQL-скрипта БД
    def rule_generating_class(self, cls):
        # класс является генерируемым, если он неабстрактный,
        # его экземпляры помещаются в хранилище и не являются типом данных.
        """
        generating_class(Cls):-
            element(Cls, 'Class'), \+stereotype(Cls, 'abstract'),
            stereotype(Cls, 'OODB'), \+internal_only(Cls).
        """
    def rule_class_parent(self, cls, nparent):
        # nparent - ближайший неабстрактный предок класса cls
        """
        class_parent(Cls, NParent):-parent(Cls, NParent),
            \+stereotype(NParent, 'abstract').
        class_parent(Cls, GNParent):-parent(Cls, NParent),
            stereotype(NParent, 'abstract'), class_parent(NParent, GNParent).
        """
    . . . . . .
class SQLTranslator(Translator, RulesMixing):
    #модуль генерирования SLQ-скрипта базы данных (БД)
    def genClass(self, cls):
        # генератор SQL-скрипта таблицы БД для класса cls
        answer = [] # код представляется списком строк
        if cls in self.generated:  # если уже сгенерирован,
            return answer          # то ничего не делать
        for _, parent in self.query('class_parent', (cls, '#')):
            # сначала сгенерировать все неабстрактные предки
        . . . . . .
        name=self.getName(cls)     # получить имя класса
        doc=cls.getDocumentation() # документация о классе
        if doc:                    # документация непуста?
            answer.append('/*\n%s\n*/' % doc) # сгенерировать комментарий
        attribs = self.genSchema(cls) # сгенерировать атрибуты таблицы БД
        if not self.isEmpty(attribs):
            answer.append("CREATE TABLE %s (" % name)
            answer.append(attribs) # атрибуты таблицы
            answer.append(")%s;" % self.getTableType(cls))
        else:
            print "The class has no attributes, generation skipped."
        self.addFact("oodb_table('%s', '%s')" % (cls.getId(), name))
        # факт, указывающий на соответствие класса сгенерированной таблице БД
        self.generated.append(cls)
        return answer # ответ - сгенерированный код
```

Заключение

В работе рассмотрена методика реализации подхода к созданию информационных систем на основе архитектуры, управляемой моделированием (Model Driven Architecture — MDA). Информационная система синтезируется по формальному описанию, представленному в UML, под управлением формального представления свойств программной платформы, на которой эта ИС реализуется. Процедуры преобразования построены с использованием методов искусственного интеллекта, а именно систем, основанных на формализованных знаниях. Формальное описание модели ИС обрабатывается набором модулей анализа и генерирования программного кода, в результате чего, в частности, порождается непротиворечивый программный код базовых подсистем ИС (SQL-скрипт создания базы данных, объекты приложения и вариант пользовательского интерфейса).

Одной из исходных целей MDA является ослабление зависимости ИС от программных технологий, при помощи которых реализуется эта ИС. По требованию заказчика путем реализации дополнительных программных модулей возможно генерирование версии ИС на совершенно отличной платформе и при помощи других технологий.

Список литературы

[1] Буч Г., Рамбо Дж., Джекобсон А. UML: Руководство пользователя. М.: ДМК, 2001. 432 с.

[2] OMG Model Driven Architecture. http://www.omg.org/mda/

[3] Object Management Group. http://www.omg.org/

[4] UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf

[5] Extensible Markup Language (XML). http://www.w3.org/XML/

[6] Python Programming Language. http://www.python.org/

Поступила в редакцию 18 марта 2005 г.