КОМПЬЮТЕРНЫЕ НАУКИ

УДК 519.7

ПРЕОБРАЗОВАНИЕ МАШИННЫХ ПРОГРАММ В АЛГОРИТМЫ, ПРЕДСТАВЛЕННЫЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

ВОДОЛАЖСКИЙ А.С., ДЮБКО Г.Ф.

Рассматриваются проблемы reengineering'а машинного кода. Предлагается описание системы, осуществляющей преобразование 32- битных файлов, исполняемых под управлением операционной системы Windows, в логические схемы.

1. Введение

Проблема декодирования машинных программ и представления их на языках высокого уровня уже давно привлекает внимание специалистов компьютерных наук. Такое декодирование получило название "reengineering". Это понятие можно дословно перевести как "преобразование" или "обратное проектирование". Это процесс приведения некоторого объекта или структуры к их исходному виду, к схеме, по которой ранее были построены этот объект или структура. Reengineering охватывает достаточно обширную область. Он используется для анализа сложных информационных систем, бизнес-процессов, структур данных. Например, системы визуального проектирования баз данных (БД), описывающие их структуру в виде некоторых диаграмм, а затем по ним формирующие реальную базу, позволяют производить обратный процесс – для готовой БД создавать соответствующую ей схему с подробным описанием таблиц, связей между ними и т.д. Также reengineering нашел применение и в области анализа программных систем. Он используется для преобразования программных модулей в некоторую форму, соответствующую, к примеру, исходному тексту или блок-схеме алгоритма. Существующие на данный момент системы частично решают эту задачу. В целом же reengineering программ является предметом дальнейших исследований. Необходимо отметить, что существует еще одно понятие – "reverse engineering", которое также подразумевает процесс обратного проектирования, но уже с учетом семантических особенностей анализируемой системы. В данном контексте приведенные выше понятия близки по смыслу, поэтому не будем проводить четких границ между ними, а в дальнейшем будем пользоваться первым термином.

2. Способы представления программ

Приступая к процессу обратного проектирования, первое, на что нужно обратить внимание, – способ представления программы, так как для разных случаев применяются различные методы анализа. Рассмотрим три варианта представления: исходный листинг, байт-код и исполняемый код. Первый вариант наиболее прост для восприятия. Он дает возможность четко проследить логическую структуру программы, определить наиболее важные (ключевые) участки, используемые переменные, внешние процедуры, визуально проследить процесс модификации переменных. Однако автоматизация процесса анализа программы, представленной исходным текстом, сопряжена с некоторыми трудностями. По сложности данная задача сравнима с задачей построения компилятора. Для преобразования программы необходимо полностью проанализировать ее содержимое (произвести трансляцию) и определить конструкции языка – переменные, условия, циклы и т.д. Затем, анализируя полученные конструкции, можно построить схему программы. На данный момент существуют системы, позволяющие производить подобный вид reengineering'а (Rational Rose).

Намного проще производить обратное проектирование, если программа представлена в некотором байт-коде. Она чем-то напоминает обычную программу в машинных кодах. Отличие заключается в том, что каждому коду соответствует не машинная команда, а некоторая конструкция языка высокого уровня. Таким образом, необходимость предварительной трансляции во внутренний формат отпадает и можно сразу приступать к построению схемы программы (или восстановлению исходного текста). Следует упомянуть, что в байт-коде обычно представляются программы на интерпретируемых языках, таких как Java.

3. Определение компилятора

Рассмотренные подходы четко ориентированы на язык программирования, на котором были первоначально написаны программы. В процессе анализа последних приходится оперировать с уже готовыми, построенными логическими конструкциями, что существенно облегчает задачу. Сложнее обстоит дело с программами в машинных кодах. Прежде, чем приступать к анализу, необходимо определить, какой компилятор использовался для создания того или иного модуля и, если возможно, язык программирования. Рассмотрим некоторые приемы (сразу надо отметить, что работа с исполняемыми модулями специфична для разных операционных систем, мы будем рассматривать особенности ОС Windows и процессоров семейства Intel x86). Компилятор Microsoft Visual C++ 6.0, например, записывает в заголовок создаваемого файла блок длиной около 60 байт, содержащий незначимую информацию ("мусор") и заканчивающийся последовательностью "Rich". Другие версии Visual C++ можно определить по содержащейся внутри программы строке вида "Microsoft Visual C++ Runtime

Library", а также перечню имен функций и сообщений об ошибках из стандартной библиотеки Microsoft C++. Подобным способом можно определить и программы, созданные в среде Borland Delphi или Builder – они обычно содержат строки "Borland Corp", "Delphi" или "C++Builder". Кроме того, компилятор помещает в exe-файл информацию об используемых объектах и типах переменных (не известно, правда, с какой целью). Для продуктов Borland характерно использование типов данных вроде TwinControl или TForm. Кроме того, в заголовке файла присутствует поле, в которое записывается номер версии компоновщика файла. Это поле не является обязательным, однако, как показывает практика, компиляторы действительно заносят туда свой номер версии. Это позволяет определить не только язык программирования, но также версию компилятора и, соответственно, учесть особенности создаваемого программного кода.

4. Логическая структура программы

При работе с модулями машинного кода необходим низкоуровневый анализ, в основе которого лежит анализ машинных команд. Предлагаемый подход (о котором в дальнейшем пойдет речь) к решению задачи reengineering'а исполняемых модулей заключается в трансляции машинного кода и сопоставлении одиночным командам или группам команд процессора конструкций некоторого языка высокого уровня, на котором, возможно, была написана программа.

Программа представляет собой последовательность команд, выполняющих заданное действие. Все команды процессора делятся на несколько групп: арифметические, логические, работы с битами, передачи управления и т.д. Кроме того, всю систему команд можно разбить на два класса. В один класс войдут команды передачи управления, в другой – все остальные. Последовательность команд из второго класса представляет собой прямолинейный участок кода, который обычно заканчивается командой перехода (первый класс). Таким образом, если проанализировать весь модуль и определить все возможные ветвления, то можно представить программу в виде некоторой структуры. Эта структура будет состоять из набора логических блоков, представляющих собой прямолинейные участки, и завершающих эти блоки команд передачи управления (условного, безусловного перехода, вызова подпрограмм, прерываний), отражающих связи между выделенными структурными единицами. В дальнейшем под понятием "блок" будем понимать именно такие элементы (рис. 1), т.е. суть метода заключается в исследовании всего кода, определении ветвлений в программе и создании базы данных с информацией о каждом блоке в структуре (детальнее в [1]). Эта информация будет использоваться в дальнейшем анализе.

5. Получение исходного текста

Теперь рассмотрим непосредственно процесс перехода от низкоуровневого представления программы к высокоуровневому (к представлению на некотором языке программирования). Каждая программа состоит из набора данных и набора команд, оперирующих этими данными. Поэтому очень важным этапом в процессе обратного проектирования является определение используемых в программе переменных, их типов (целые, вещественные, массивы), областей видимости (глобальные, локальные). Выделение конкретных переменных из общего набора данных позволит частично определить логику программы и подготовит почву для следующего этапа – определения конструкций языка.

В процессе предварительного анализа, рассмотренного ранее, программа разбивается на логические блоки, представляющие собой линейные участки кода – последовательности команд, следующих одна за другой. Эти команды оперируют некоторыми локальными или глобальными данными. Следует более тщательно рассмотреть эти два понятия. Под глобальными переменными понимаются данные, расположенные в некоторой общей области памяти, к которой можно получить доступ из любого участка программы. В данном случае, в качестве такой области выступает сегмент данных программы. Понятие же локальных переменных несколько отличается от общепринятого. В рассматриваемом контексте оно обозначает не только переменные, имеющие ограниченную область видимости, например, внутри функции, тем более что такие элементы языка как "функция" еще не распознаны на текущем этапе. Формально каждый блок оперирует некоторыми данными, т.е. какие-то переменные являются для него входными, хранящими исходные значения, какие-то – выходными, содержащими результат выполнения блока, а некоторые – рабочими. Таким образом, под "локальными" на данном этапе понимаются переменные логического блока. Определение таких переменных является достаточно важным шагом в процессе reengineering'а – позволяет установить более глубокие связи между блоками, основанные уже не только на передаче управления, но и на одновременно используемых переменных: ведь выходные переменные одного блока являются одновременно входными другого (см. рис.1).

Рис. 1. Логический блок

После определения переменных можно перейти, собственно, к формированию конструкций языка. При этом следует рассмотреть два подхода к анализу: "от частного к целому" и, наоборот, "от целого к частному". Первый подход подразумевает формирование сначала простых операций – арифметических, логических, присваивания, вызовов функций; затем объединение их в более крупные, например, циклы, затем в еще более крупные – функции и т.д. до тех пор, пока не будет составлен

полный листинг программы. Второй подход предполагает противоположные действия. В самом начале в программе выделяются несколько крупных участков, соответствующих, например, нескольким функциям. Затем анализируется каждый из них. Выделяются условные операторы, циклы и т.д. до определения простых операций (сложение, умножение). Преимущество данного подхода: анализ ведется в пределах конкретного участка, границы которого могут быть определены.

Наиболее просто формируются арифметические и логические операции, обычно они содержатся внутри одного логического блока (прямолинейного участка кода). Кроме того, таким операциям соответствует одна, максимум две машинные команды. Это позволяет без особого труда определять достаточно большое число конструкций языка.

Но все далеко не так просто, как может показаться на первый взгляд. Процесс reengineering'а машинного кода имеет множество подводных камней. Начнем с того, что каждому элементу языка высокого уровня соответствует несколько (очень редко одна) команд процессора, т.е. нужно достоверно знать, какая конкретная последовательность команд отвечает конкретному элементу языка. А если эти же самые команды расположены в другом порядке, но результат дают тот же самый? Возникает неоднозначность. В принципе, одни и те же команды могут трактоваться по-разному: например, оператор присваивания может быть не просто оператором присваивания, а частью конструкции цикла. Рассмотрим строку "for(i=a+b; i<c; i++)". В процессе анализа оператор "i=a+b" может быть выделен как самостоятельная конструкция языка. На самом же деле (и это можно выяснить при более глубоком и тщательном анализе) он является частью цикла "for". Это же касается и "i++". В то же время, значительно отличающиеся участки, в которых команды переставлены местами, задействованы разные регистры, могут, по сути, давать один и тот же результат и соответствовать одной и той же языковой конструкции. Таким образом, жесткая привязка к набору машинных инструкций не даст желаемого эффекта. Нужно использовать семантические методы анализа.

Кроме того, что каждой операции языка высокого уровня соответствует несколько команд ассемблера, они могут быть еще и разнесены по нескольким логическим блокам. Это естественно – для формирования простейшего условного оператора необходимо проанализировать как минимум два блока. А если это сложная конструкция, например, вложенные циклы? Сложность задачи возрастает в несколько раз. В итоге, необходимо одновременно рассматривать несколько блоков и выделять составные структурные элементы языка.

Также существенно затрудняет работу наличие так называемых "косвенных" вызовов. Типичными примерами являются создание процессов, потоков, обращение к функциям через специальные индексные таблицы (подобным способом вызываются функции операционной системы). В этом случае адрес процедуры либо рассчитывается, либо передается в качестве параметра другой процедуре, либо берется из какой-нибудь переменной. Для того чтобы определить реальный адрес, необходимо проследить процесс изменения локальных переменных в нескольких блоках. А количество этих блоков заранее неизвестно, так как сложно сказать, с какого именно момента нужно начинать отслеживание изменений. Промоделировать же работу всей программы и, следовательно, пронаблюдать за всеми изменениями практически невозможно.

Блок оптимизации кода в современных компиляторах достаточно хорошо продуман и успешно справляется со своей задачей. Это создает еще проблему для reengineering'а – оптимизированный код гораздо сложнее анализировать. Некоторые конструкции упрощены, некоторые заменены более эффективными. Соответственно, объем исходных данных для работы уменьшается. Для получаемого в результате текста программы характерны избыточность и низкая степень соответствия изначальному исходному коду.

6. Пример

А теперь продемонстрируем действие изложенного выше подхода на примере. На рис. 2 приведена небольшая программа на языке C++, осуществляющая сортировку массива по возрастанию методом "пузырька". В программе присутствуют все основные элементы языка (функция, вложенные циклы, условный оператор и т.д.). В качестве исходных данных функция "sort" получает адрес массива и количество содержащихся в нем элементов.

```
void sort(int *mas, int length)
{
    int i, j, k;

    for(i=0; i<length-1; i++)
        for(j=i+1; j<length; j++)
            if(mas[i]>mas[j])
                        {
                k=mas[i];
                mas[i]=mas[j];
                mas[j]=k;
            }
}
```

Рис. 2. Текст программы

.....	локальные переменные
ebp-8	
ebp-4	
ebp+0	старое значение указателя EBP
ebp+4	адрес выхода из функции
ebp+8	передаваемые параметры
ebp+12	(располагаются в обратном
.....	порядке)

Рис. 3. Состояние стека при вызове функции

Приступим непосредственно к анализу. Следует отметить, что для компиляции программы на рис. 2 был использован транслятор из пакета Visual C++

6.0. Авторами данной статьи была разработана программа логического анализа машинного кода, создающая структуру исполняемых файлов и отображающая ее в виде последовательности логических блоков. С помощью этой программы был получен результат, приведенный на рис. 4 (машинный код разбивается на 11 блоков, приводятся адреса, мнемоники и операнды команд). Рассмотрим работу каждого блока в отдельности.

В блоках 1 и 11 производится настройка стека и подготовка специальных регистров процессора для работы с локальными переменными. Следует уде-

```
; Блок №1
00401000:    push   ebp                        ;начальная инициализация
00401001:    mov    ebp,esp
00401003:    sub    esp,0C
00401006:    push   esi
00401007:    mov    [dword ptr ebp-04],0       ;инициализация некоторой
0040100E:    jmp    00401019                   ;локальной переменной var1
; Блок №2
00401010:    mov    eax,[dword ptr ebp-04]
00401013:    add    eax,01
00401016:    mov    [dword ptr ebp-04],eax     ;увеличение var1 на 1
; Блок №3
00401019:    mov    ecx,[dword ptr ebp+0C]     ;сравнение var1 с параметром
0040101C:    sub    ecx,01                     ;length, уменьшенным на 1
0040101F:    cmp    [dword ptr ebp-04],ecx     ;переход на блок №11,
00401022:    jnl    00401082                   ;если var1>=length-1
; Блок №4
00401024:    mov    edx,[dword ptr ebp-04]     ;инициализация второй локальной
00401027:    add    edx,01                     ;переменной var2 значением
0040102A:    mov    [dword ptr ebp-08],edx     ;var1+1
0040102D:    jmp    00401038
; Блок №5
0040102F:    mov    eax,[dword ptr ebp-08]
00401032:    add    eax,01
00401035:    mov    [dword ptr ebp-08],eax     ;увеличение var2 на 1
; Блок №6
00401038:    mov    ecx,[dword ptr ebp-08]     ;сравнение var2 со значением
0040103B:    cmp    ecx,[dword ptr ebp+0C]     ;length, переход на блок №10,
0040103E:    jnl    00401080                   ;если var2>=length
; Блок №7
00401040:    mov    edx,[dword ptr ebp-04]     ;сравнение двух элементов
00401043:    mov    eax,[dword ptr ebp+08]     ;массива mas (адрес массива
00401046:    mov    ecx,[dword ptr ebp-08]     ;передается в качестве параметра
00401049:    mov    esi,[dword ptr ebp+08]     ;функции), индексы которых
0040104C:    mov    edx,[dword ptr eax+edx*4] ;задаются переменными var1 и
0040104F:    cmp    edx,[dword ptr esi+ecx*4] ;var2, переход на блок №9,
00401052:    jng    0040107E                   ;если mas[var1]<=mas[var2]
; Блок №8
00401054:    mov    eax,[dword ptr ebp-04]     ;перестановка местами элементов
00401057:    mov    ecx,[dword ptr ebp+08]     ;массива mas с использованием
0040105A:    mov    edx,[dword ptr ecx+eax*4] ;третьей локальной переменной
0040105D:    mov    [dword ptr ebp-0C],edx
00401060:    mov    eax,[dword ptr ebp-04]
00401063:    mov    ecx,[dword ptr ebp+08]
00401066:    mov    edx,[dword ptr ebp-08]
00401069:    mov    esi,[dword ptr ebp+08]
0040106C:    mov    edx,[dword ptr esi+edx*4]
0040106F:    mov    [dword ptr ecx+eax*4],edx
00401072:    mov    eax,[dword ptr ebp-08]
00401075:    mov    ecx,[dword ptr ebp+08]
00401078:    mov    edx,[dword ptr ebp-0C]
0040107B:    mov    [dword ptr ecx+eax*4],edx
; Блок №9
0040107E:    jmp    0040102F                   ;переход на блок №5
; Блок №10
00401080:    jmp    00401010                   ;переход на блок №2
; Блок №11
00401082:    pop    esi                        ;выход из функции
00401083:    mov    esp,ebp
00401085:    pop    ebp
00401086:    retn
```

Рис. 4. Машинный код

лить особое внимание принципам организации стека и распределения памяти под временные данные. Эти принципы подробно описываются в соответствующей литературе по C++. Поэтому мы лишь вкратце остановимся на них. Все локальные переменные функции (в том числе и параметры) хранятся в стеке. Доступ к ним осуществляется через регистр EBP, как показано на рис. 3 (все поля имеют размер 4 байта, поэтому значение смещения, прибавляемого к EBP, изменяется с шагом 4 и записано в десятичном виде). Таким образом, адрес массива будет находиться по смещению EBP+12, а его размер – по смещению EBP+8. Кроме того, в первом блоке происходит инициализация некоторой локальной переменной (EBP-4), назовем ее var1, т.е. можно записать строку вида: “int var1=0;”. Затем управление передается на 3-й блок.

В блоке №3 значение по смещению EBP+12, уменьшенное на единицу, сравнивается с уже определенной переменной var1. А EBP+12, как было установлено ранее, - это один из параметров функции. Блок заканчивается командой сравнения. В случае выполнения условия происходит переход на последний блок, т.е. выход из функции. Исходя из этого, можно записать следующую строку кода: “if(var1<length-1) {...}”. Вместо “...” в дальнейшем будет записан код, определенный в блоках 4-10. В 4-м блоке появляется вторая локальная переменная (по смещению EBP-8), назовем ее var2: “int var2=var1+1;”. Далее следует передача управления на блок №6, который напоминает только что описанный блок №3. Запишем: “if(var2<length) {...}”.

Следующие два блока, как видно из рис. 4, содержат множество команд загрузки регистров значениями из соответствующих ячеек памяти. Суть седьмого блока сводится к сравнению двух элементов переданного в качестве параметра массива, индексы элементов задаются переменными var1 и var2: “if(mas[var1]>mas[var2])”. Если условие удовлетворяется, выполняется блок №8, в котором фактически происходит перестановка элементов местами (с участием третьей переменной var3). Осталось рассмотреть действие блоков №2 и 5. Они практически идентичны, за тем лишь исключением, что один увеличивает на единицу переменную var1, а другой – var2. Объединяя все обрывки в единое целое, получаем исходный текст программы на C++, приведенный на рис. 5.

Полученный исходный код не соответствует изначальному тексту программы. Это естественно, ведь стопроцентного соответствия получить невозможно, об этом уже упоминалось ранее. Однако можно немного преобразовать полученный код, чтобы добиться более качественных результатов. Так, можно убрать операторы “goto” и, объединив их с последующим условием “if”, получить циклы типа “while”. Перед этими циклами стоят команды инициализации цикловых переменных, а в конце циклов - увеличения переменных. На основании этого получаем две конструкции “for”. Осталось заменить переменные var1, var2, var3 на i, j, k, и получится исходная программа.

```
void sort(int *mas, int length)
{
    int var1=0;

m1: if(var1<length-1)
    {
        int var2=var1+1;
m2:     if(var2<length)
        {
            if(mas[var1]>mas[var2])
            {
                int var3=mas[var1];
                mas[var1]=mas[var2];
                mas[var2]=var3;
            }
            var2++; goto m2;
        }
        var1++; goto m1;
    }
}
```

Рис. 5. Полученный текст

Исходя из изложенного выше, можно выделить несколько правил определения элементов языка высокого уровня:

1) арифметические и логические команды ассемблера преобразуются в соответствующие арифметические и логические операторы языка;

2) логические блоки, заканчивающиеся командами условного перехода, соответствуют конструкции “if”. Внутри таких блоков обычно находятся команды сравнения. Тот участок программы, на который указывает переход, должен относиться к первой части условного оператора, а блоки, расположенные ниже команды перехода (получающие управление в случае невыполнения условия), – к конструкции “else”;

3) если участок кода, находящийся внутри оператора “if”, завершается командой передачи управления и эта команда указывает на данный условный оператор (т.е. на начало блока сравнения), то вся конструкция представляет собой цикл типа “while” (см. выше преобразования программы на листинге 3). Таким же образом, если некоторый участок программы завершается командами сравнения и перехода на начало этого участка, можно получить цикл с постусловием;

4) если, кроме описанного в предыдущем пункте, перед выполнением цикла инициализировалась, а в конце блока цикла – изменялась участвующая в сравнении переменная, то формируется оператор “for”.

7. Заключение

Рассмотренный подход к анализу программных систем может использоваться для получения логических схем программ, определения рабочих переменных, преобразования программ в более удобный для анализа формат, выделения логических

конструкций, характерных для языков высокого уровня. Семантический подход при reengineering'е состоит в применении базы знаний (БЗ) для соответствующих преобразований. База знаний может быть представлена в форме трехуровневой модели знаний (грамматика – таблица решений – исчисление предикатов) [2]. Верхним уровнем БЗ являются правила принятия решений, выраженные в форме Хорновских дизъюнктов. Для приведенного выше примера в БЗ можно включить правила формирования оператора цикла и условного оператора в виде:

а) Оператор Цикла ($f(i,m_0,m_1,m_2,n_0,n_1,n_2,a)$): $Init(n_0,i,m)$, $Body(n_1,n_2,i,a)$, $Cmp(n_2,i,m1)$, $Inc(n_2,i,m_2)$, $Jmp(n_2,n_1)$.

б) Условный Оператор ($f(i,m_2,n_0,n_1,n_2,a)$): $Cmp(n_0,i,m_2)$, $Jmp(n_0,n_2)$, $Body(n_1,n_2,a)$.

Оператор цикла состоит из блоков инициализации некоторой рабочей переменной (Init), сравнения (Cmp), увеличения (Inc), передачи управления (Jmp) и тела цикла (Body). Условный оператор включает в себя конструкции сравнения, передачи управления и тело оператора. Обозначения:

– $f(b_1,...,b_k)$ – некоторая функция от k переменных;

– n_0, n_1, n_2 – номера блоков (соответственно – начальный, промежуточный, конечный);

– i – рабочая переменная;

– m_0, m_1, m_2 – значения рабочей переменной (начальное, величина изменения, конечное);

– a – тело цикла или условного оператора.

Литература: 1. *Водолажский А.С.* Преобразование программ в удобную для логического анализа форму // Сборник научных трудов по материалам 4-го Международного молодежного форума "Радиоэлектроника и молодежь в XXI веке". Х.: ХНУРЭ, 2000. С. 333-334. 2. *Дюбко Г.Ф., Валенда Н.А., Водолажский А.С.* Семантический подход к автоматическому анализу программ / Сборник научных трудов по материалам 5-й Международной конференции "Теория и техника передачи, приема и обработки информации". Х.: ХНУРЭ, 1999. С. 360-362.

Поступила в редколлегию 25.03.2002

Рецензент: д-р техн. наук, проф. Авраменко В.П

Водолажский Алексей Сергеевич, студент ХНУРЭ. Научные интересы: защита информации. Адрес: Украина, 61115, Харьков, ул. 2-Пятилетки, 2-В, кв. 144, тел. 94-71-29.

Дюбко Геннадий Федорович, канд. техн. наук, профессор кафедры ПО ЭВМ ХНУРЭ. Научные интересы: формальные системы. Адрес: Украина, 61140, Харьков, пр. Гагарина, 38, кв. 70, тел. сл. 40-94-46, дом. 27-13-26.

УДК 519.62

ДОПОЛНИТЕЛЬНЫЕ КОМПОНЕНТЫ ИНФОРМАЦИОННОЙ МОДЕЛИ ГИС

ЛЯХОВЕЦ С.В., ЧЕТВЕРИКОВ Г.Г.

Приводятся дополнительные компоненты информационной модели ГИС. Эти компоненты предназначены для решения проблемы управления проектами сложных природно-технических комплексов. Рассматриваются компоненты для хранения топологических данных, иерархии трубопроводов, произвольной группировки событий, а также все необходимые таблицы баз данных для хранения данных компонентов модели.

Введение

На протяжении жизненного цикла любого трубопровода из различных источников собираются большие количества информации в разных форматах. Возникает необходимость создания информационной модели для организации и управления данными, процессами сбора и использования информации. Информационная модель трубопровода – это согласованная, хорошо управляемая, легко доступная совокупность всей информации о трубопроводе, среде и эксплуатационной истории. Ее независимость от конкретных программ является существенным фактором для обеспечения гибкости и возможности многократного использования одних и тех же данных. Наработки в этой области можно изучить в источниках [15, 16].

Информационная модель предоставляет следующие преимущества:

1. Качественное использование и доступ к данным: уменьшение избыточности данных; многократное использование данных; оптимизация связей между отделениями компании.

2. Делает более эффективным принятие решений: постепенное сокращение работы при соборе и форматировании данных; расширенная аналитическая среда; улучшение формулировки рационального решения, построения сценария и тестирования; улучшение инструментальных средств документирования; способность включать различные варианты в среду принятия решения.

3. Поддерживает приложения для улучшения эксплуатационной эффективности: поддержка отчетов и разрешений; оперативное получение информации по району работ; автоматизированная картография – создание подписей объектов и полосы карты непосредственно из базы данных; планирование и маршрутизация для службы и запросов технического обслуживания; генерация карт размещения, дающих ориентировочную информацию для облегчения обнаружения находящегося под землей оборудования; интеграция с документацией на оборудование позволяет обеспечивать доступ к новейшим документам по техническому обслуживанию, действующему оборудованию и методикам.