Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования
«Уральский федеральный университет имени первого Президента России Б.Н. Ельцина»

Институт математики и компьютерных наук

Кафедра высокопроизводительных компьютерных технологий

ВИЗУАЛЬНАЯ ОТЛАДКА ПОСТРОЕНИЯ ГРАФА ПРОГРАММЫ В ТРАНСЛЯТОРЕ СИ99

«Допущена к защите»

Зав.кафедрой
А.В. Созыкин

_____

«___»_____2014 г.

Квалификационная работа
на соискание степени бакалавра наук
по направлению
«Математика, компьютерные науки»
студентки гр. КН-401,
Анненковой Ольги Геннадьевны

Научный руководитель
Бахтерев М.О., м.н.с., ассистент КВКТ

Екатеринбург
2014

РЕФЕРАТ

Анненкова О.Г., ВИЗУАЛЬНАЯ ОТЛАДКА ПОСТРОЕНИЯ ГРАФА ПРОГРАММЫ В ТРАНСЛЯТОРЕ СИ99, квалификационная работа на степень бакалавра: стр. 24, рис. 4, библ. 3 назв., приложений 3.

Ключевые слова: ВИЗУАЛИЗАЦИЯ, ПРЕДСТАВЛЕНИЕ ДАННЫХ, ОТЛАДКА, ГРАФ ПРОГРАММЫ, СЕМАНТИЧЕСКИЙ ДАГ

Объект исследования - семантические графы, генерируемые транслятором Си99. Цель работы - разработка визуального представления для отладки построения графа программы. В рамках работы исследуется возможности информативного визуального представления динамически изменяющихся структур данных, возникающих в процессе работы над специфической моделью вычислений. В ходе исследования разрабатывается оригинальная методика отладки, полученные в результате работы программы применяются в действующем проекте разработки компилятора Си99 для мультиклеточных процессоров, основанном на новом подходе к построению компиляторов путем событийно-управляемой трансляции.

Содержание

ВВЕДЕНИЕ

Специфика семантического анализа исследуемой системы трансляции состоит в применении набора форм к потоку определенных синтаксических инструкций. Форма регистрируется в контексте вывода, подставляется в выражение и преобразует его, последовательность таких операций с множеством форм постепенно формирует семантический даг программы. Для отладки этого процесса необходимо анализировать состояние дерева областей вывода после применения группы форм. Таким образом, для относительно простого синтаксического выражения генерируется порядка нескольких тысяч строк отладочной информации. Работа с такими объемами данных в настоящее время не является сложной вычислительной задачей, однако, такой материал становится неприемлемым для человеческого восприятия, необходимо представить исследуемую структуру в более удобном в использовании формате. Задача разработки средств компьютерной визуализации для представления и анализа информации и структур данных является востребованной в различных областях научных исследований и в настоящее время разработано множество методик и алгоритмов (см. обзорные публикации [1], [2]).

Проблемы отладки программ повсеместно решаются пошаговым выполнением в отладчике, т.е. трассировкой по точкам останова, или анализом трассы программы, полученной добавлением вывода отладочных сообщений. Эти способы предоставляют универсальные возможности локализации ошибок и проверки корректности, но позволяют анализировать только статичные наборы значений некоторых частных характеристик, которые не способны показать общую глобальную динамику процесса исполнения, теряются детали, структурные особенности и объектные взаимосвязи. В сложных системах для анализа больших объемов информации требуются специальные инструменты мониторинга.

В данной работе исследуются возможности представления отладочного дампа в интерактивном виде, рассматриваемые подходы: трансляция в гипертекст и генерация графической анимации. В тексте работы описываются алгоритмы моделирования визуального представления, анализируются особенности, достоинства и слабые стороны используемых методов и технологий с точки зрения сложности создания кода, информативности и эффективности использования.

Широкие возможности для представления различного рода информации предоставляют веб-технологии, но идея использования среды обозревателя веб-страниц в области анализа структур данных и отладки программ в настоящее время не находит упоминаний в литературе, поэтому такой подход можно считать оригинальным. Использование этого метода располагает рядом преимуществ: простотой динамической генерации, интерактивными и навигационными средствами. Однако, предполагается, что гипертекст не способен отразить некоторых особенностей исследуемой динамической стркуктуры, например, интерес может представлять слияние двух контекстов на вершине стека для объединения левой и правой частей бинарной инструкции, поэтому будет рассмотрен также и графический подход. Статические графические изображения не способны передать изменения такого рода, необходимы более гибкие средства визуализации, поэтому следует генерировать анимационную графику, для создания которой будет использоваться готовая библиотека визуализации динамических графов, в тексте будут описаны ее функции и основные алгоритмы работы.

1. Описание формата данных

Данные состоят из последовательности записей данных стека и форм, применяющихся на текущем шаге. Состояние стека представлено в виде набора ассоциативных массивов (map-структур из списка пар ключ-значение), отличающихся уникальными адресами. Значение по некоторому ключу также может быть адресом некоторого массива или семантического дага пустого или с набором именованных узлов, где узел может ссылаться на предшествующие. Каждая форма записывается в общем универсальном виде и в расширенном конкретными атомами: операциями, идентификаторами, константами и др. Ключами массивов являются списки атомов. Информация на последующем шаге может отличаться от предыдущего добавлением/удалением ассоциативных массивов, добавлениями пар ключ-значение, изменением значения по некоторому ключу, добавлением/удалением дага или добавлением новых его узлов.

Пример map-структуры со ссылкой на семантический граф.

```
map: 0xbb6ef0
  key(0): ('00.1."^"; '00.6."LNKCNT")
  val(0): 1
  key(0): ('00.1."*"; '00.5."FORMS")
  val(0): ()
  key(0): ('00.1."*"; '00.3."DAG")
  val(0): D:0xb7a580

  dag: 0xb7a580
  (
    .TEnv n0  ('00.5."float");
    .T  n1  ('00.3."LOC"; T:1);
    .S  n2  ('02.1."a"; n1)
  )
```

Пример формы.

```
(
  .FIn  n0  ();
  .Nth  n1  (n0; (1));
  .FEnv n2  ('00.10."MAKESYMBOL");
  .FPut n3  (0; (('00.10."IDENTIFIER"); ('00.4."TYPE")); n2);
  .TEnv n4  ('00.5."float");
  .FOut n5  (0; ((('00.4."TYPE"); n4)));
  .FOut n6  (0; ((('00.10."IDENTIFIER"); n1)));
  .E  n7  ('00.3."ENV"; ('00.4."this"));
  .Go n8  (n7)
)
```

Таким образом, разрабатываемая программа должна по тексту описанного формата уметь генерировать визуальное представление, выделяющее происходящие изменения в структуре в удобном наглядном виде.

2. Трансляция в гипертекст

Транслятор будет выделять особенности, представляющие наибольший интерес, путем изменения стиля текста. Алгоритм работы: разбор структуры, поиск различий с данными предыдущего шага и вставка тегов, перестановка с учётом вложенности, для форм выделение расширенных строк, добавление обозначений участков со статусом "done", выделение дополнительных сообщений, добавление css-стилей.

При разборе стека понадобится подсчет табуляций, для map нужно получить список вложенности, расположение в виде номеров строк и соответствие ключам значений, для дагов - вложенность и список узлов (см. алгоритм 1).

Алгоритм 1

function PARSE($Stack$)
  $addr := ''''; tab := -1; numb := 0; i := 0; tabstack := [[addr, tab]];$
  for all $str \leftarrow Stack$ **do**
    if $Map \leftarrow str$ **then**
      $MapsL[addr] := [numb, i-1]; numb := i; t0 := tab;$
      while $tabcount(str) <= t0$ **do** $t \leftarrow tabstack; t0 := t[1];$
      end while
      $parent \leftarrow tabstack[-1]; Maps[parent].add(Map);$
      $tabstack.add([addr := Map, tab := tabcount(str)])$
    else if $Key \leftarrow str$ **then**
      $s \leftarrow stack; i{+}{+}; MapsH[Map][str] := s;$
    else if $Dag \leftarrow str$ **then**
      $i{+}{+}; s \leftarrow Stack;$
      $prefix := substr(s, 0, -1); dag := \emptyset;$
      while $s \neq (prefix + '')''$ **do** $dag.add(s); s \leftarrow Stack;$
      end while
      $Dags[addr][Dag] := dag;$
    end if
    $i{+}{+};$
  end for
  return $[Maps, MapsL, MapsH, Dags];$
end function

Изменениями стека будем считать только добавленные новые map структуры, графы, новые ключи или измененные значения по ключу, для этого сравниваем разобранный стек на текущем шаге с предыдущим (см. алгоритм 2).

Алгоритм 2

```
1:  PrsOld := ∅;
2:  function DIFF(Stack)
3:      Prs := PARSE(Stack); MapsL := ∅; MapsH := ∅; Dags := ∅;
4:      (MapsL1, MapsH1, Dags1) ← Prs; (MapsL0, MapsH0, Dags0) ← PrsOld;
5:      for all Map ∈ MapsL1 do
6:          if Map ∉ MapsL0 then MapsL.add(Map);
7:          end if
8:      end for
9:      for all Map ∈ MapsH1 do
10:         for all (Key, Val) ∈ MapsH1[Map] do
11:             if Map ∉ MapsH0 or Key ∉ MapsH0[Map] or MapsH0[Map][Key] ≠ Val then
12:                 MapsH[Map][Key] := Val;
13:             end if
14:         end for
15:     end for
16:     for all (Map, Dag) ∈ Dags1 do
17:         if (Map, Dag) ∉ Dags0 then Dags[Map][Dag] := Dags1[Map][Dag];
18:         end if
19:     end for
20:     PrsOld := Prs; return [MapsL, MapsH, Dags];
21: end function
```

Далее рекурсивно переставляется порядок (алгоритм 3): описание map ставится ближе к ссылке, для этого при нахождении адреса проверяется является ли ссылка дочерней для текущей map для избежания циклов.

Алгоритм 3

function ORDERCHANGE($Stack, Map, Maps, MapsL$)
  $Stack1 := \emptyset$; $Stack1.add(MapsL[Map][0])$
  $n := tabcount(Stack[MapsL[Map][0]])$;
  for $(i := (MapsL[Map][0] + 1); i < MapsL[Map][0]; i{+}{+})$ **do**
    $Stack1.add(Stack[i])$;
    if $Addr \leftarrow Stack[i]$ and $Addr \in Maps[Map]$ **then**
      $Stack1$:=ORDERCHANGE($Stack, Addr, Maps, MapsL$);
    end if
  end for
  return $Stack1$;
end function

Для форм изменения считаются построчным сравнением. Результат получается после добавления тегов и стилей, пример: рис. 1.

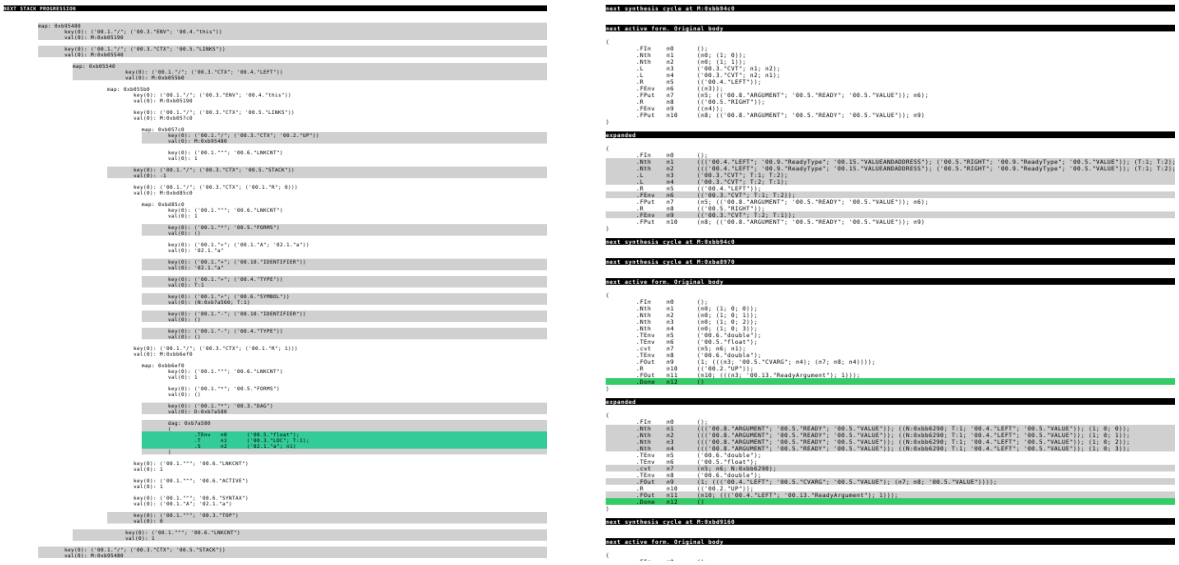


Рис. 1: а) Дерево областей вывода. б) Формы.

В результате получилось простое и эффективное средство отладки. Отрицательная сторона состоит в том, что количество текста не уменьшилось, никак не отражено связывание графов, циклы ссылок и удаленные части структуры. Использованный метод достаточно тривиален и может применяться в самых различных задачах. Преимуществом также стоит отметить скорость реализации.

3. Графическое представление

Для создания анимационной графики будет использована библиотека динамической визуализации графов UbiGraph, сборка поставляется в виде сервера поддерживающего формат XML-RPC[1], клиенты могут быть написаны на языках, реализующих поддержку протокола, таким образом, методы вызываются HTTP-POST запросами. Среди методов добавление/удаление вершин и ребер по идентификатору и назначение атрибутов. Для высокой производительности рендеринга трехмерной сцены используется OpenGL, Pthreads и ряд алгоритмов моделирования динамики, плавно адаптирующихся к онлайн изменениям (см. [3]).

В основе идеи алгоритмов библиотеки лежит метафора физической частицы: вершина v_i представляется точечной частицей пространства, координаты x_i расчитываются по физическим формулам. Алогритм вычисления механики движения получается из преобразований и огрублений уравнения Эйлера-Лагранжа с единичными массами и учётом силы трения и уравнений для кинетической и потенциальной энергии (E_T и E_V) по формулам: 3.1, 3.2, 3.3,

$$\left[\frac{d}{dt}\frac{\partial}{\partial \dot{x}_i} - \frac{\partial}{\partial x_i}\right](E_T - E_V) = 0 \tag{3.1}$$

$$E_T = \sum_{v_i \in V} \frac{1}{2}||\dot{x}_i||^2 \tag{3.2}$$

$$E_V = \sum_{(v_i,v_j)\in E} \frac{1}{2}K||x_i - x_j||^2 + \sum_{v_i,v_j \in V, v_i \neq v_j} \frac{f_0}{\epsilon + ||x_i - x_j||} \tag{3.3}$$

где V - множество вершин,

E - множество рёбер,

K - коэффициент упругости,

f_0, ϵ - константы.

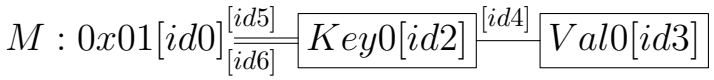
[1]Extensible Markup Language Remote Procedure Call - протокол XML-вызова удалённых процедур

Решение находится численными методами в графе с рандомизированными огрублениями, что позволяет ускорить вычисления и гладко сходится к более точному решению. Огрубление графа получается алгоритмом быстрого динамического связывания, использующим случайные приоритеты и граф зависимостей ребер.

Для визуализации исследуемой структуры будем сводить ассоциативный массив к графу путём последовательного соединения ключей, первый и последний соединим с дополнительным узлом именованным адресом, образуя петлю, в целях упрощения представления, ссылки будем обозначать пунктирными рёбрами. Сделанное упрощение способствует возможности хранения и поиска идентификаторов узлов и ребер, которое требуется для проведения вставки и удаления компонент (см. рис. 2).

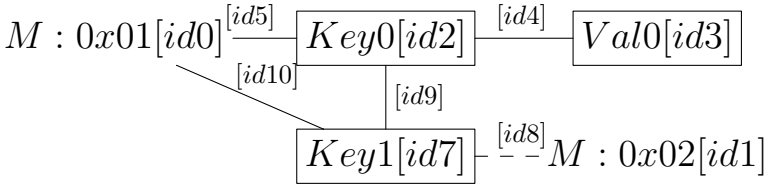
В целом алгоритм следующий: разбор структуры очередного шага, добавление новых и удаление лишних компонент, временная задержка для рендеринга и просмотра. Потребуется функция получения нового идентификатора ($newId()$) и ассоциативный массив для хранения идентификаторов (ID).

Для вставки нового ключа в map на место i будем создавать вершину для ключа, вершину для значения, если оно не является ссылкой, в противном случае ищем нужный идентификатор, ребро между ними, далее будем рассматривать случаи вставки в пустой map, в конец, в середину и в начало, т.к. получаются различные связывания (см. алгоритм 4).



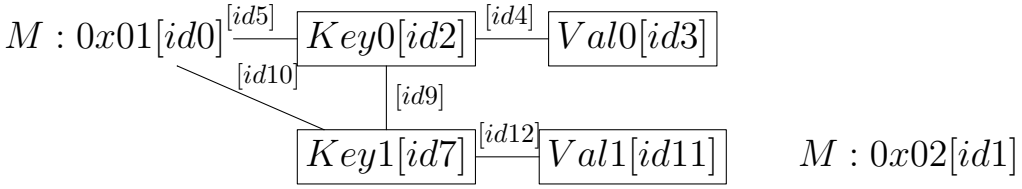

```
ID = {
  'M:0x01':[id0,[id2,id3,id4,id5],id6];
  'M:0x02':[id1]
}
```

Insert Key:




```
ID = {
  'M:0x01':[id0,[id2,id3,id4,id5],[id7,id1,id8,id9],id10];
  'M:0x02':[id1]
}
```

Replace Val:




```
ID = {
  'M:0x01':[id0,[id2,id3,id4,id5],[id7,id11,id12,id9],id10];
  'M:0x02':[id1]
}
```

Рис. 2: Пример работы с идентификаторами.

Алгоритм 4

```
1:  function INSERTKEY(M, Key, Val, i)
2:      ADDVERTEX(id0 ← newId());
3:      if Ref ← Val then id1 := ID[Ref][0]
4:      else ADDVERTEX(id1 ← newId());
5:      end if
6:      ADDEDGE(id2 ← newId(), id1, id0);
7:      if |ID[M]| == 1 then
8:          ADDEDGE(id3 ← newId(), id0, ID[M][0]); ID[M].add([id0, id1, id2, id3]);
9:          ADDEDGE(id4 ← newId(), ID[M][0], id0); ID[M].add(id4);
10:     else if |ID[M]| == i + 2 then
11:         ADDEDGE(id3 ← newId(), id0, ID[M][i][0]);
12:         ADDEDGE(id4 ← newId(), ID[M][0], id0);
13:         REMOVEEDGE(ID[M][i + 1]);
14:         ID[M][i + 1] = [id0, id1, id2, id3]; ID[M].add(id4);
15:     else
16:         if i > 0 then
17:             ADDEDGE(id3 ← newId(), id0, ID[M][i][0]);
18:         else
19:             ADDEDGE(id3 ← newId(), id0, ID[M][0]);
20:         end if
21:         ADDEDGE(id4 ← newId(), ID[M][i + 1][0], id0);
22:         REMOVEEDGE(ID[M][i + 1][3]); ID[M][i + 1][3] := id4;
23:         ID[M].insert(i + 1, [id0, id1, id2, id3]);
24:     end if
25: end function
```

В случае замены значения по ключу с номером i следует удалить ребро, связывающее ключ со значением, вершину соответствующую значению, если оно не содержит ссылку, добавить новый ключ, если такого узла еще нет, и ребро к нему (см. алгоритм 5).

Для изменений map понадобятся еще созданиие пустой структуры и удаление всех компонент за исключением ссылок (см. алгоритм 6).

Алгоритм 5

```
1: function REPLACEVAL(M, Val, i)
2:     L := ID[M][i + 1]; f := 0;
3:     REMOVEEDGE(L[2]);
4:     for all Ref ∈ ID do
5:         if L[1] == ID[Ref][0] then f := 1;
6:         end if
7:     end for
8:     if not f then REMOVEVERTEX(L[1]);
9:     end if
10:    if Ref ← Val then
11:        id1 := ID[Ref][0];
12:    else
13:        ADDVERTEX(id1 ← newId());
14:    end if
15:    ADDEDGE(id2 ← newId(), id1, L[0]);
16:    ID[M][i + 1] = [L[0], id1, id2, L[3]];
17: end function
```

Алгоритм 6

```
1: function ADDMAP(M)
2:     ADDVERTEX(id0 ← newId()); ID[M].add(id1);
3: end function
4: function REMOVEMAP(M)
5:     REMOVEVERTEX(ID[M][0]);
6:     for all |L| > 1, L ∈ ID[M] do
7:         REMOVEVERTEX(L[0]); f := 0;
8:         for all Ref ∈ ID do
9:             if L[1] == ID[Ref][0] then f := 1;
10:            end if
11:        end for
12:        if not f then REMOVEVERTEX(L[1]);
13:        end if
14:    end for
15:    ID.remove(M);
16: end function
```

Даг будем хранить простым списком вершин (к адресу приписывается адрес родительского map из-за возможного адреса nil), т.к. граф будет

удаляться только полностью, а при удалении узлов будут также удалены инцидентные ребра, при создании узлов добавляем дополнительные ребра для ссылок (см. алгоритм 7).

Алгоритм 7

```
 1: function ADDDAG(M, D, Dags)
 2:     ADDVERTEX(id0 ← newId()); ID[M + D].add(id1);
 3:     prev := id0;
 4:     for all node ∈ Dags[M][D] do
 5:         ADDVERTEX(id1 ← newId());
 6:         for all intRef ← node do
 7:             ADDEDGE(newId(), id1, ID[Ref][1 + intRef])
 8:         end for
 9:         ADDEDGE(newId(), id1, prev); prev := id1;
10:     end for
11:     if prev ≠ id0 then
12:         ADDEDGE(newId(), prev, id0)
13:     end if
14: end function
15: function REMOVEDAG(M, D)
16:     for all id ∈ ID[M + D] do REMOVEVERTEX(id)
17:     end for
18:     ID.remove(M + D);
19: end function
```

Тогда основной алгоритм изменения компонент имеет следующий вид: см. алгоритм 8. Порядок следования ключей сохраняется, чтобы не хранить соответствий с идентификаторами. В реализации добавлены также стили для всех вершин и ребер и режимы отображения подписей.

Поскольку используемая библиотека специализируется на онлайн изменениях динамическая анимация получается простой расстановкой временных задержек.

Алгоритм 8 Функция перехода к следующему состоянию.

```
1:  MapsOld = ∅; DagsOld = ∅;
2:  function NEXTSTEP(Stack)
3:      (MapsNew, DagsNew) ← PARSE(Stack)
4:      for all M ∈ MapsNew\MapsOld do ADDMAP(M);
5:      end for
6:      for all M ∈ MapsNew do
7:          for all D ∈ DagsNew[M] do
8:              if M ∉ MapsOld or D ∉ DagsOld[M] then ADDDAG(M, D, DagsNew);
9:              end if
10:         end for
11:     end for
12:     for all M ∈ MapsNew do
13:         i := 0;
14:         for all (Key, Val) ∈ M do
15:             if M ∉ MapsOld or Key ∉ MapsOld[M] then INSERTKEY(M, Key, Val, i);
16:             else if (Key, Val) ∉ MapsOld[M] then REPLACEVAL(M, Val, i);
17:             end if; i++;
18:         end for
19:     end for
20:     for all M ∈ MapsOld\MapsNew do REMOVEMAP(M);
21:     end for
22:     for all M ∈ MapsOld do
23:         for all D ∈ DagsOld[M] do
24:             if M ∉ MapsNew or D ∉ DagsNew[M] then REMOVEDAG(M, D);
25:             end if
26:         end for
27:     end for
28:     MapsOld := MapsNew; DagsOld := DagsNew;
29: end function
```

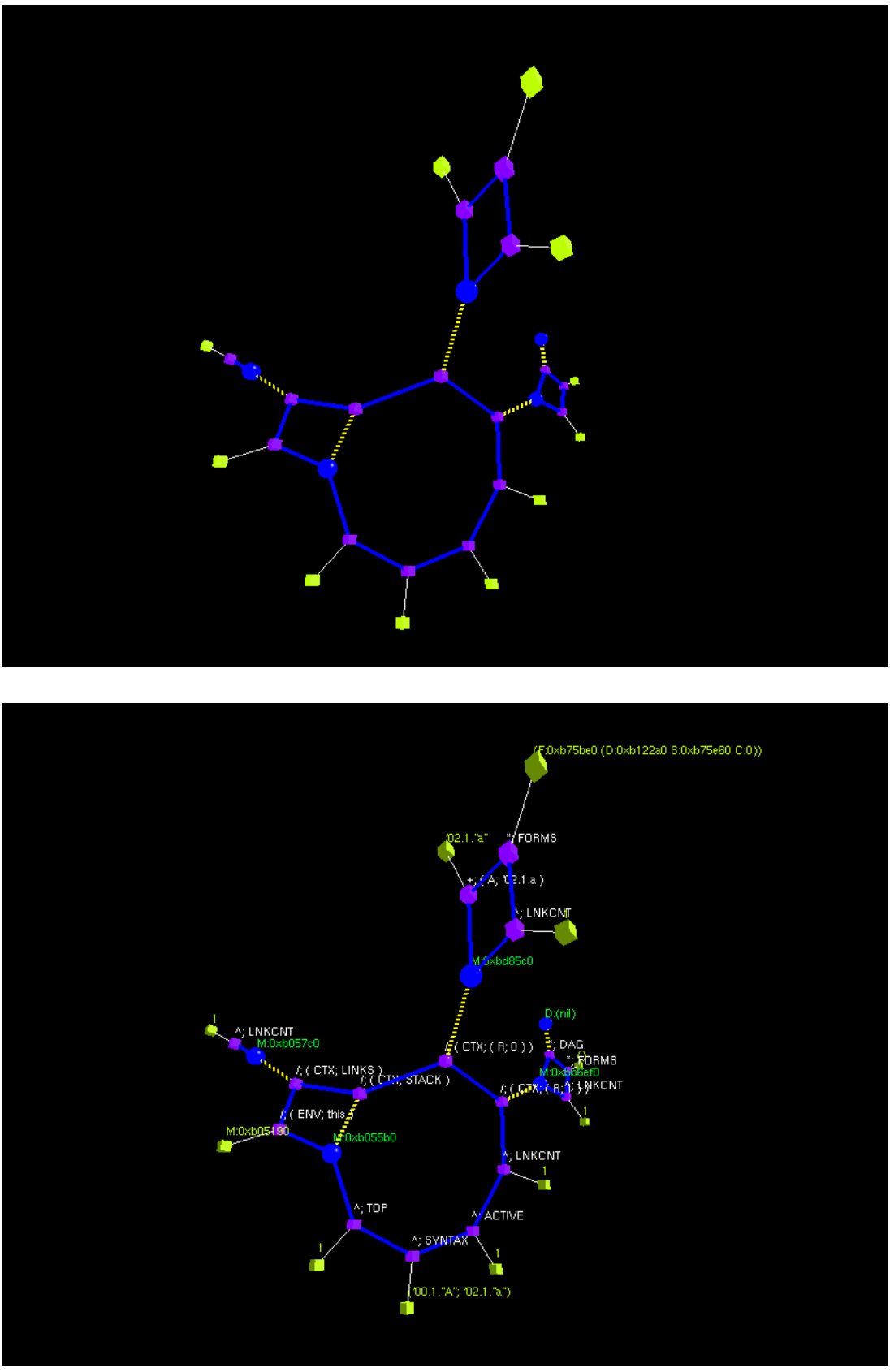


Рис. 3: Пример простой модели.

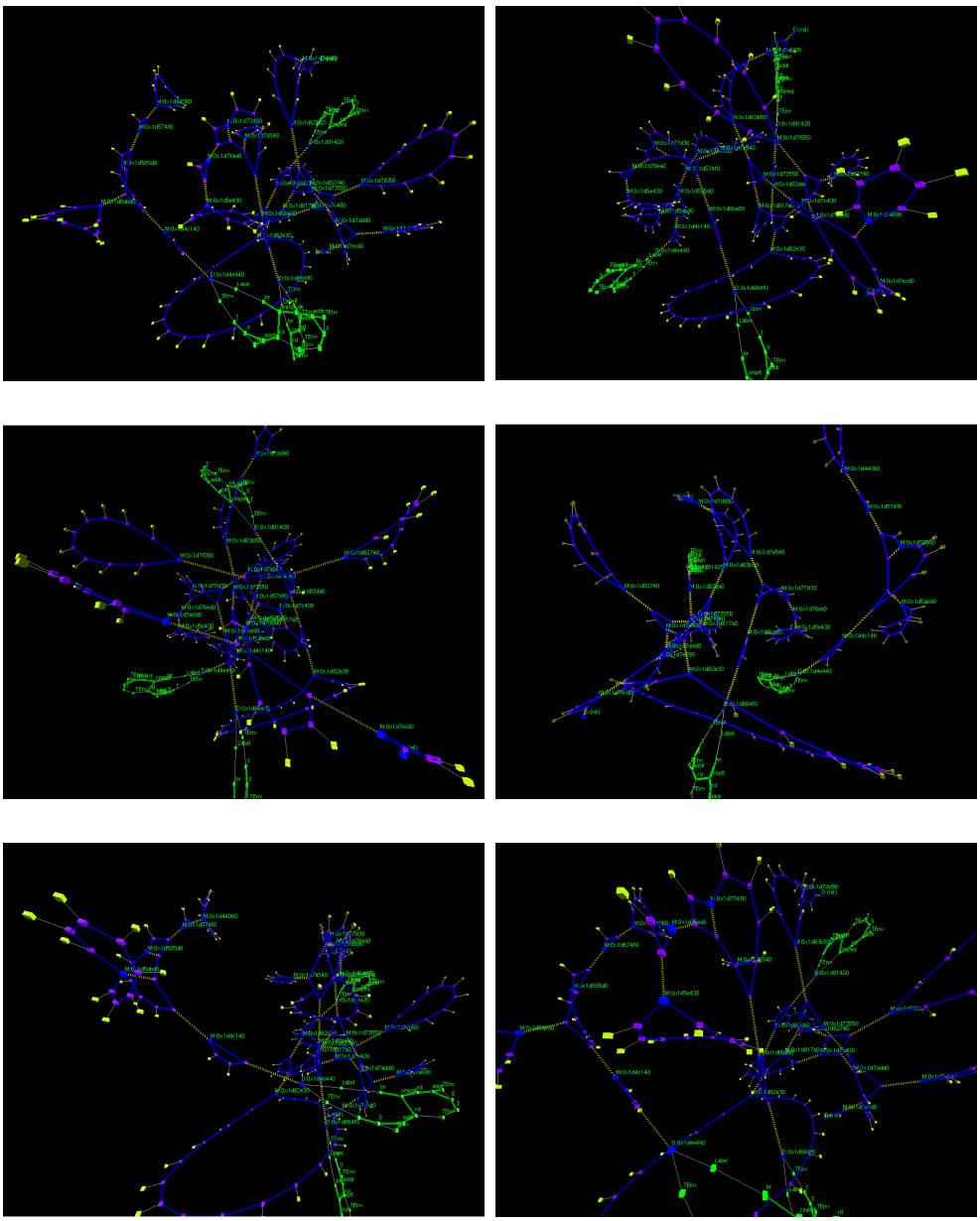


Рис. 4: Обзор сложной структуры с разных сторон.

В результате получился генератор графической анимации, который показывает общую наглядную картину исследуемого процесса, можно анализировать связность и ход выращивания графов, эффективность использования снижают трудности работы с подписями и навигацией. В данном конкретном случае метод применен довольно быстро и просто, но не стоит использовать его для подобной задачи с более сложными данными, т.к. может быть трудной работа с динамическими структурами необходимыми для хранения идентификаторов.

ЗАКЛЮЧЕНИЕ

Современные технологии предполагают гибкую работу со сложными динамически формирующимися и взаимодействующими объектами, поэтому можно отметить востребованность решения задач подобного рода. В процессе работы над созданием средств отладки особенной модели вычислений были применены разные подходы к визуальному представлению семантических данных. Интерактивные возможности гипертекста оказались эффективным пластичным средством в области визуализации структур и были успешно применены для анализа добавлений стека. Идея создания графической анимации в задаче визуализации для отладки может выглядеть достаточно неуместной, трудноосуществимой и требующей длительной разработки, но в данной работе показана возможность быстрого и простого решения проблемы, которое стало возможно благодаря современным вычислительным возможностям. Хотя полученное средство и не обладает высокой эффективностью, оно помогает получить цельное представление и отразить динамику. В результате разработаны алгоритмы, которые в некоторой мере решают поставленную задачу и находят реальное практическое применение.

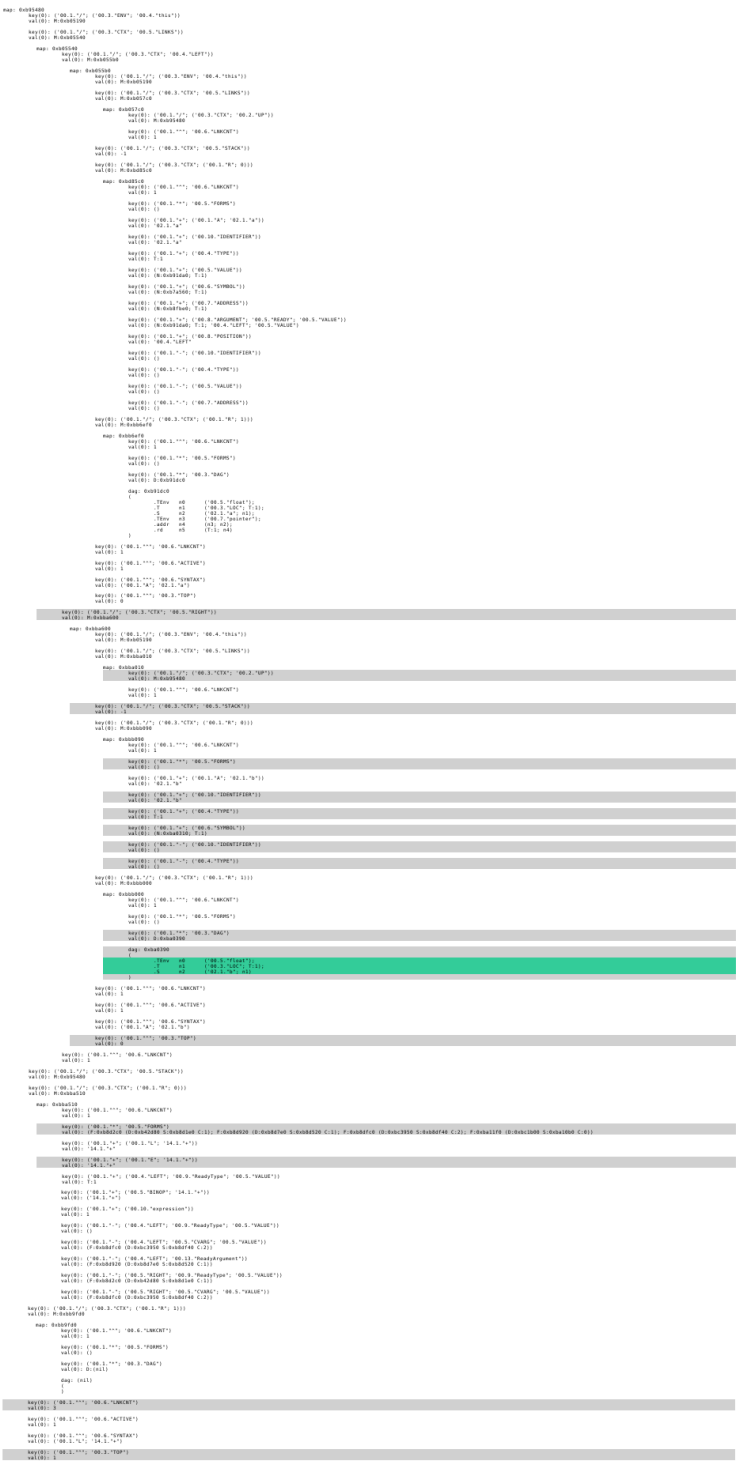
ССЫЛКИ И ЛИТЕРАТУРА

1. Herman I., Melancon G., Marshall M.S., Graph visualization and navigation in information visualization: a Survey // Visualization and Computer Graphics vol. 6, no. 1, 2000, pp. 24-43.

2. Авербух В. Л., Разработка средств компьютерной визуализации для научных исследований // Труды Первой международной конференции «Трехмерная визуализация научной, технической и социальной реальности. Кластерные технологии моделирования», Том I, электронная публикация, УДК 004.92-94, г. Ижевск, 2009 г., стр. 8-11.

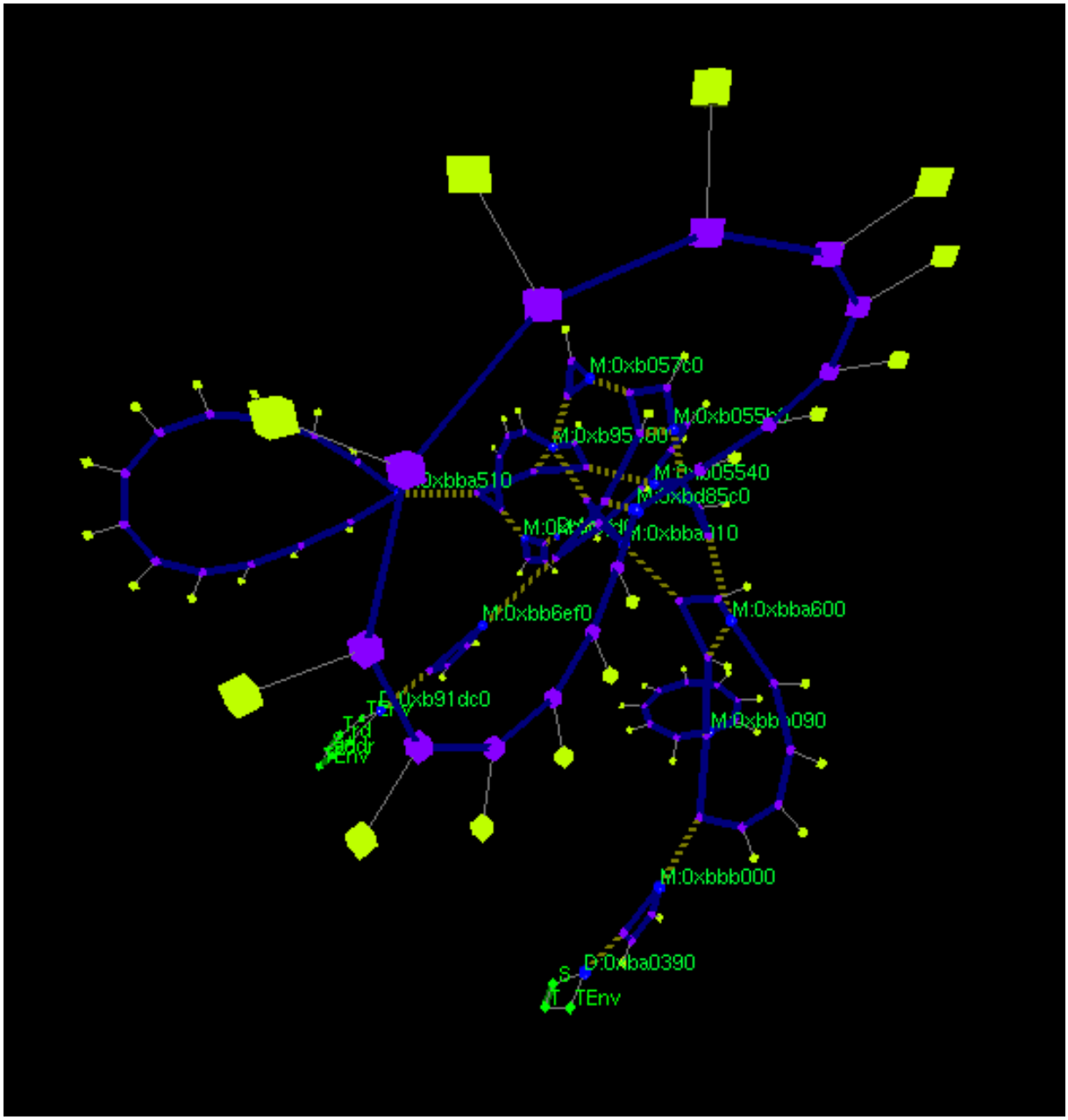
3. Todd L. Veldhuizen, Dynamic Multilevel Graph Visualization. // Electrical and Computer Engineering, Canada, 2007, pp. 1-20.

Сравнение.



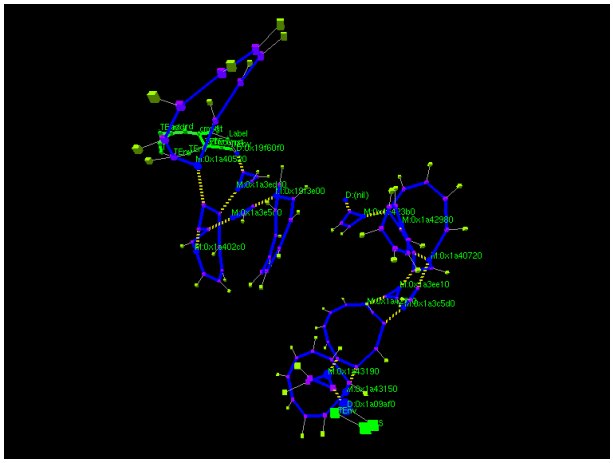
а) Гипертекст.

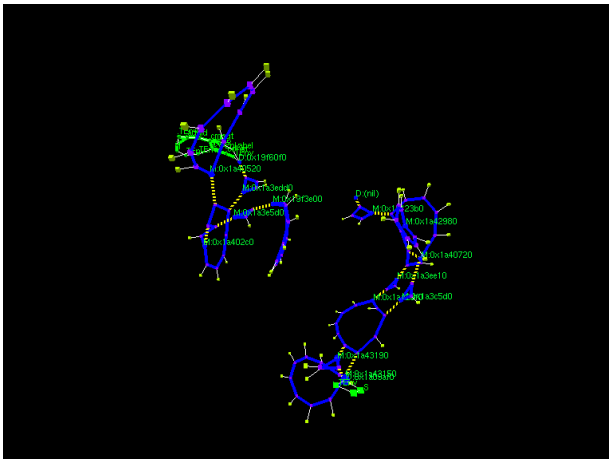
Сравнение.

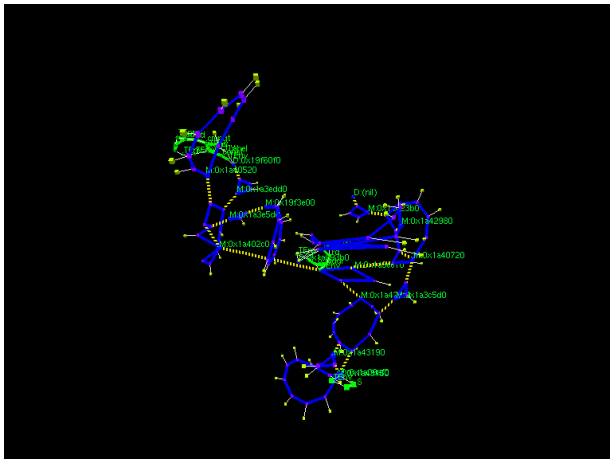


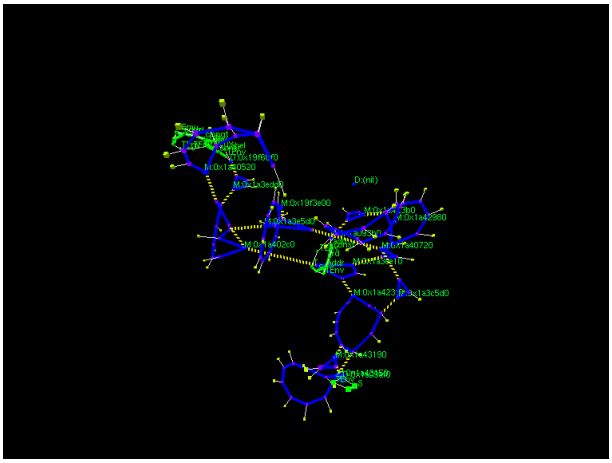
б) Графика.

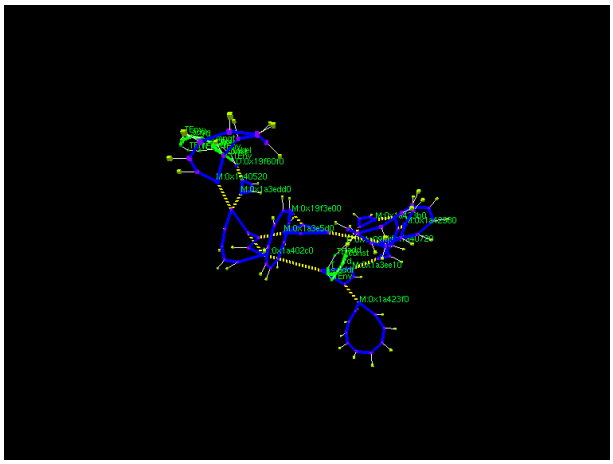
Связывание.

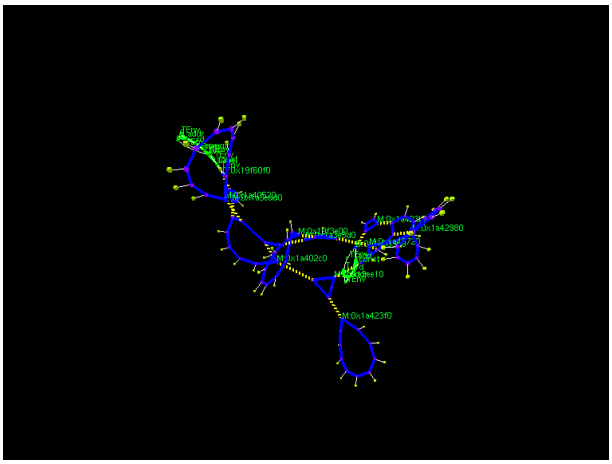












Удаления.