Новосибирский Государственный университет

Механико-математический факультет

кафедра программирования

Запреев Иван Сергеевич

Дипломная работа

“Анализ разумной организации информационных потоков в Оберон-программах”

Научный руководитель
профессор
_____И.В. Поттосин

г. Новосибирск, 2000г.

Оглавление:

1. Введение.

Существуют различные критерии добротности программных продуктов. Среди них такие показатели , как безопасность , устойчивость , понимаемость , тестируемость , хорошая организованность информационных потоков и потоков управления , а так- же многие другие .

Остановимся на следующей формулировке добротности программ , которая дается в статье [1] : Добротность в ней определяется, как " разумная , рациональная , не переусложненная организация информационных потоков управления и разумное построение вычислений ". Суженное таким образом понятие добротности программного продукта , уже не касается таких критериев , как , например , соответствие его запросам заказчика , но тем не менее, предполагая хорошую внутреннюю организованность программы , мы можем рассчитывать и на удовлетворение других требований .

С другой стороны, понимая таким образом добротность, мы все еще имеем множество различных критериев, таких как количественные , генетические , структурные и прагматические критерии.

Количественные критерии - это те, в основе которых лежит вычисление некоторых количественных характеристик программы .Такие критерии в основном позволят лишь сравнивать характеристики аналогичных программ . К таким методам относится , например , оценка управляющего графа программы (этот критерий еще известен , как метод "Совершенства" Холстеда) . Но также существуют и методы , позволяющие дать "абсолютные" оценки добротности.

Генетические критерии основаны на доверии некоторым заведомо хорошим технологиям. То есть если при разработке программы использовалась некоторая технология, которая считается добротной в некотором смысле, то и сам продукт тоже является в этом смысле добротным . При таком подходе гарантируется хорошее соответствие спецификациям разрабатываемого продукта , что само по себе является некоторым существенным критерием добротности .

Структурные критерии, характеризуют структурную добротность программы, т.е. иными словами добротной , является хорошо структурированная программа (под этим понимается хорошая организация процедур и классов).

Прагматические критерии определяют наличие "излишеств" допущенных при написании программ. Иными словами, они требуют что бы программа

соответствовала некоторой своей формально выделяемой цели (например, достижение некоторого конечного состояния переменных или подтверждаемые программным текстом потоки управления , информационные потоки и т.п.). Подробнее на этом виде критериев мы остановимся в следующем пункте дипломной работы, которая как раз и посвящена реализации одного из прагматических критериев добротности программ .

2. Постановка задачи.

Как уже было отмечено, прагматические критерии проверяют соответствие программы некоторой своей формально выделенной цели .

Все множество прагматических критериев можно разгруппировать по нескольким видам требований:

1. Целевая направленность - Здесь цель - это достижение некоторого множества допустимых конечных состояний памяти .

2. Структурная целесообразность - Требуется поддерживать видимую структуру управления .

3. Оправданная выстроенность вычислений - Нужно , чтобы операторы , необходимые для вычисления , оправдано размещались в вычислительном процессе .

4. Вычислительная неизбыточность - В этом случае целью является выработка значений переменных, которые мы формально определяем , как результаты программы .

5. Разумная организация информационных потоков - Цель такой организации информационных потоков - это поддержание обнаруженных информационных потоков.

Добротность же информационных потоков, в свою очередь , разделяется на **регулярность** (отсутствие неоправданной перепутанности информационных потоков) и **подтвержденность** (что означает , что любая переменная всегда будет инициализирована перед своим использованием).

Задачей данной дипломной работы является написание программы - анализатора , которая анализирует добротность информационных потоков в Оберон-программах на предмет их **регулярности** , поэтому , в соответствии со статьей [1] , ниже будет приведено точное определение этого вида добротности ; но прежде рассмотрим несколько определений и модель

анализируемой программы , в смысле которой в работе [1] и дается критерий регулярности информационных потоков .

Итак, во-первых для определения добротности информационных потоков используется модель линейных схем , когда программа представляется в виде линейной последовательности операторов S1,S2,..Sn , и для каждого Si известны множества A(Si)- множество его аргументов ,R(Si)- множество результатов и R'(Si)- множество обязательных результатов или как еще говорят сильно результатное множество . При этом R'(Si) - подмножество R(Si) , содержащее только те результаты, которые обязательно будут выработаны при выполнении оператора Si .

Рассмотрим линейную схему T = S1...Sn .

Тогда по определению говорят, что Sj зависит от Si (i<j) , если :

a) существует такая переменная x, что x принадлежит R(Si), x принадлежит A(Sj), x не принадлежит R'(Si+1...Sj-1). Что означает , что значение x выработанное Si доходит до Sj , где потом используется .

b) cСуществует такая переменная x, что x принадлежит R(Si), x принадлежит R(Sj)\R'(Sj), x не принадлежит R'(Si+1...Sj-1). Что означает , что значение выработанное Si , доходит до Sj , а тот его меняет или нет. При этом

если значение не меняется, то тогда его можно считать псевдоаргументом Sj.

Информационным графом I для последовательности операторов T , называется ориентированный граф , вершины которого - это операторы из T , а из вершины k существует дуга в j вершину, если Sj зависит от Sk .

Информационным потоком I(Sk) оператора Sk называется подграф графа I , включающий все вершины из которых существуют пути в k вершину , в том числе и саму вершину Sk .

Реализацией I(Sk) назовем подпоследовательность T(Sk)= Sp,...,Si , где p - это минимальный номер вершины в I(Sk).

Информационные потоки I(Sk) I(Sj) называются независимыми, если они не содержат общих вершин, и пересекающимися, если они содержат общие вершины, но ни один из них не является подграфом другого.

Усеченной реализацией I(Sn) по отношению к пересекающемуся с ним I(Sm) называется такая последовательность операторов T(Sn:Sm), для которой оператор с минимальным в подпоследовательности номером берется без учета номеров общих для них вершин (операторов).

И так, основываясь на описанных выше определениях, в работе [1] вводится следующая точная формулировка критерия добротности информационных потоков с точки зрения их регулярности:

КРИТЕРИЙ:

Информационные потоки в программе с точки зрения регулярности добротны, если как для нее, так и для всех ее линейных фрагментов любого уровня вложенности справедливо следующее:

a) Реализация любого информационного потока I(S) не содержит операторов, относящихся к независимому по отношению к I(S) информационному потоку .

b) Усеченная реализация I(Sn) по отношению к пересекающемуся с ним I(Sm)не содержит операторов, для которых соответствующие вершины принадлежат I(Sm) и не принадлежат I(Sn).

Из формулировки критерия ясно, что нарушение пункта a) сразу же приводит к неоправданному переплетению информационных потоков, что же касается пункта b) то он тоже контролирует отсутствие неоправданного переплетения информационных потоков, но менее тривиальным образом.

Пример:

Пусть есть последовательность операторов ABCDEF при этом они связаны следующим образом:

Оператор F зависит от A,D ;

Оператор E зависит от D,B ;

Оператор D зависит от C ;

Тогда очевидно, что при последовательности вычислений ABCDEF, мы сначала, начиная с вычисления A, нужного для F , бросаем их и начинаем вычисления B, нужные для E. При этом заметим, что T(F:E)=ABCDEF содержит B и E, не принадлежащие информационному потоку I(F).

Что же касается реального применения данного критерия добротности, то нужно отметить, что само по себе отсутствие переплетения информационных потоков (их регулярность)означает соответствие текста программы реальным информационным связям, что в свою очередь повышает понимаемость программы, и в дополнении к этому важно отметить, что последовательные программы с добротными с точки зрения регулярности информационными потоками без дополнительной перестановки операторов можно естественно

разбивать на последовательности вычислений, обрабатываемых параллельными процессами, которые в свою очередь могут выполняться на параллельно работающих процессорах.

3. Потоковый процессор.
(Структура внутреннего языка.)

Итак, критерий регулярности информационных потоков предполагает наличие некоторых знаний о самой программе. Так для каждого оператора Si необходимо знать множество его аргументов A(Si), множество его результатов R(Si), и множество его сильных результатов R'(Si). Таким образом анализатор регулярности информационных потоков естественно рассматривать, как "программный процессор" работающий после другого программного процессора - Потокового процессора, который реализует абстрактную интерпретацию программы в смысле Кузо.

Данный потоковый процессор является частью разрабатываемого статического анализатора OSA (Шелехов В.И. Куксенко С.В.) (Oberon-2 / Modula-2 Static Analyser) для программ на смеси языков Оберон-2 / Модула-2 в окружении системы программирования XDS для OS/2 , Windows 95/NT и SunSparc .

Потоковый процессор является препроцессором статического анализатора семантических ошибок периода исполнения . И в этом контексте , разрабатываемый анализатор, является некоторым промежуточным звеном анализа программы между двумя этими процессорами .

Как программа, потоковый процессор реализован на языках Модула-2/Оберон-2, и представляет набор взаимодействующих между собой модулей . Нужно отметить , что основным для данной работы являлся модуль Pass.ob2 , содержащий процедуры обхода строящегося в процессе работы потокового процессора атрибутированного дерева, на основе которого строилось дерево линейного представления программы , необходимое анализатору добротности информационных потоков .

Потоковый процессор реализует межпроцедурный контекстно - чувствительный анализ, и также обеспечивает аппроксимацию как обязательных, так и возможных информационных связей. Само представление информационных связей будет описано далее в виде потоковой модели программы в пунктах 3.1 - 3.4.

А теперь подробнее будет описана, модель, реализуемая потоковым

процессором:

3.1. Объекты.

Под объектом понимается объект памяти исполняемой программы. Для языков Модула -2, Оберон -2 , это переменные, процедуры и процессы. Процедура считается внешней, если вызов реализуется в анализируемой конфигурации, но тело отсутствует. Все переменные можно разделить на внешние (экспортируемые переменные и потенциально доступные во внешних процедурах) и внутренние (все остальные).

Так же различаются переменные:

1) Переменные, описанные в тексте программы.
2) Переменные памяти, существующие в памяти исполняемой программы.
3) Абстрактные переменные, принадлежащие потоковой модели программы.

При этом любая переменная памяти аппроксимируется единственной абстрактной переменной.

Абстрактные переменные в свою очередь классифицируются следующим образом:

1. Основные переменные - соответствующие переменным, описанным в программе.
2. New переменные - Генерируемые при исполнении операторов ALLOCATE, NEW.
3. Компонентные переменные - Компоненты структурных переменных - ARRAY, RECORD.
4. Предыдущие экземпляры переменных. (относится к New переменным).

Переменные генерируемые операторами NEW, ALLOCATE аппроксимируются абстрактными переменными с именами $newK, где K - номер вхождения в программе вызова генератора.

Два экземпляра переменной различаются в потоковом процессоре следующим образом : текущий экземпляр New переменной аппроксимируется абстрактной переменной $newK , а все предыдущие экземпляры , генерируемые тем же вызовом NEW , аппроксимируются другой абстрактной переменной с именем $newK' .

Между элементами массива различий не делается, и поэтому все они апроксимируются одной абстрактной переменной - обобщенным элементом массива. Имена для поля переменной записи, обобщенного элемента массива являются:

<имя переменной-записи>.<имя поля >

<имя переменной -массива>[]

3.2. Объектные выражения.

Объектным выражением является всякая языковая конструкция, обозначающая объект в тексте программы.

Для языков Модула -2 и Оберон -2 можно привести следующую запись, раскрывающую суть объектного выражения:

<объектное выражение > ::= <имя переменной> | <имя процедуры> |
ADR(<объектное выражение>) |
<объектное выражение>^ |
<объектное выражение>[<выражение>] |
<объектное выражение>.<имя поля> |
<объектное выражение>.<имя альтернативы
объединения> |
<объектное выражение>.<имя метода> |
<объектное выражение>(<тип>) |
<объектное выражение>
(<список фактических параметров>) |
CAST(<объектное выражение>,<тип>)

Покрытие объектного выражения - результат вычисления потоковым процессором конструкции <объектное выражение> есть один или несколько объектов. При этом считается, что абстрактная переменная **входит** в позиции некоторого объектного выражения, если абстрактная переменная принадлежит его покрытию.

Бывают следующие виды вхождения абстрактной переменной в объектное выражение:

1. Определяющее (когда реализуется присваивание переменной, обозначаемой объектным выражением).
2. Использующее.
3. Адресное.

3.3. Определения переменных.

Для представления информационных связей в потоковом процессоре применяются SSA формы (static single assignment form). Это позволяет обеспечить для каждого использующего вхождения переменной одно определяющее.

Например, на месте определяющего вхождения переменной X в потоковом процессоре строится определение переменной следующего вида:

<имя абстрактной переменной> (<номер определения>) [<статус>]

{ <значение> }

Статусы могут быть:

1. "d" - обязательное определение - для любого исполнения соответствующего объектного выражения произойдет присваивание переменной.
2. "p" - возможное определение - для некоторого исполнения соответствующего объектного выражения произойдет присваивание переменной.
3. "u" - определение отсутствует.

Значение вычисляется как для скалярных так и для объектных типов. Здесь под объектными понимаются типы POINTER и PROCEDURE. Для них значениями являются соответственно одна или несколько абстрактных переменных, процедур. Они называются **заместителями** определяемой переменной. Каждый заместитель имеет статус:

"d" - обязательный заместитель .

"p" - возможный заместитель .

Для скалярных же типов значением является список диапазонов значений типа.

Следует отметить, что в точках слияния нескольких ветвей программы потоковый процессор вставляет новые определения переменных через Ф функции.

В начале каждой процедуры потоковым процессором создается набор определений, называемых аргументами процедуры, для глобальных переменных и формальных параметров процедуры, имеющих в ней использующее вхождение.

Реализация от различных вызовов процедуры фиксируется набором вариантов процедуры. Если число вариантов больше одного , то для всех определений процедуры, а также для всех заместителей используется вместо одного статуса [<статус>],[<вектор статусов>].

Итак, как уже отмечалось, если есть использующее вхождение переменной, то ему соответствует единственное определяющее вхождение той же

абстрактной переменной. Это однозначно определяет информационные связи в программе. При этом для представления множества информационных связей используется структура, которую можно назвать контекстом определений, она является результатом отображения множества абстрактных переменных на соответствующие определения переменных. И таким образом для некоторого использующего вхождения переменной, определяющим будет то, которое в данный момент находится в контексте определений.

Итак, выше было дано описание потоковой модели программы, которое реализует потоковый процессор. Нужно отметить, что реально его работа заключается в построении некоторого древовидного представления программы, дальнейшего обхода и атрибутирования этого дерева. При этом контекст определений переменных строится одновременно с обходом и существует лишь только в текущей точке обхода программы. Сам потоковый процессор допускает несколько вариантов обхода дерева, которые происходят попроцедурно. Обход происходит по всем операторам тела процедуры.

При этом существуют некоторые особенности реализации. Например, цикл WHILE исходной программы интерпретируется потоковым процессором, как безусловный (бесконечный) цикл LOOP (Oberon-2 , Modula-2) с условным оператором внутри, который содержит условие завершения цикла.

Пример:

```
WHILE (УСЛОВИЕ) DO
.......
END;
```

Будет интерпретироваться следующим образом:

```
LOOP
  IF  NOT (УСЛОВИЕ) THEN
  .......
  ELSE
     EXIT;
  END;
END
```

В тоже время существует и более существенная особенность связанная с параметрами процедур. Все VAR параметры процедур заменяются на простую передачу адреса переменной, что в целом не меняет их суть.

4. Реализация алгоритма.

В связи с тем, что потоковый процессор реализован на языках Modula-2 , Oberon-2 , а построенный анализатор должен был тесно интегрироваться с исходными текстами этого процессора, то для написания анализатора добротности информационных потоков были также выбраны языки Oberon-2 , Modula-2 .

При реализации вся задача по построению данного анализатора была разбита на две подзадачи:

1) Построение линейной модели программы. Это извлечение необходимой информации от потокового процессора о тестируемой программе. То есть разбиение ее на линейные последовательности операторов , вычисление для всех операторов множеств их аргументов , результатов и сильных результатов , выделение информационных связей между операторами .

2) Анализ по построенной модели добротности информационных потоков. В соответствии с указанным ранее критерием добротности.

4.1. Построение линейной модели программы.

Эта часть задачи очень тесно переплетается с работой потокового процессора . Мною был написан модуль **analyser.ob2** на языке Oberon-2 , который содержит набор процедур, вызовы которых вставлены напрямую в исходники одного из модулей потокового процессора (модуля **Pass.ob2**), отвечающего за обход дерева внутреннего представления анализируемой программы. (Как уже упоминалось, потоковый процессор сначала строит некоторое древовидное представление программы, и потом в дальнейшем обходит его, атрибутируя, вычисляя текущий контекст , этот обход и реализует модуль Pass.ob2.)

Обход идет по процедурам , начиная с самых верхних, т.е. с тех, из которых нет вызовов других процедур, с имплементированными в анализируемой программе телами, и заканчивая теми, цепочки вызовов процедур из которых имеют максимальную длину. Таким образом, имеется возможность последовательного вычисления множеств A , R , R' для процедур, и в дальнейшем использовать их в местах соответствующих вызовов.

Итак, при обходе, реализуемом в модуле Pass.ob2, вызовами процедур модуля **analyser.ob2** строится древовидная структура программы, которая реализует линейную модель программы. В строящейся структуре каждый оператор является либо простым оператором, либо составным, и тогда он содержит

набор ветвей, которые представляют собой последовательные списки операторов, содержащихся на этих ветвях. Для оператора **if** две ветви, для **case** количество ветвей произвольное, для циклов и процедур это одна ветвь, содержащая последовательность операторов от начала цикла (процедуры) и до конца. При этом структурные операторы содержат раскрытие своих ветвей в себе.

Нужно отметить, что эта модель реализована в виде динамических списков структур, с широким использованием структурного полиморфизма, обеспеченного стандартом языка Oberon-2.

При последовательном обходе потоковым процессором тела процедуры (обход идет последовательно по всем операторам, и ветвям (для структурных операторов)) при прохождении операторов, запоминается текущий контекст переменных - контекст перед оператором, и потом, после прохождения этого оператора этот контекст сравнивается с контекстом после пройденного оператора и вычисляя разницу контекстов после и до текущего оператора мы получаем множество его результатов R. При этом в процессе обхода оператора добавляется соответствующая ему структура (элемент списка (оператор) текущей ветви) в строящуюся линейную модель процедуры, и полуученое множество R - результатов этого оператора запоминается в отвечающей ему структуре (элементе списка соответствующей ветви).

Так как контекст представляет набор SSA форм то и множество R - результатов оператора , тоже является набором SSA форм переменных. Тех, для которых появилось новое описание при прохождении оператора. Далее на основе SSA формы из множества R, вычисляется множество R' - обязательных результатов данного оператора, при этом так как потоковый процессор проводит контекстно-чувствительный анализ программы, то SSA форма содержит информацию о всех вызовах анализируемой процедуры, а значит, что строящиеся множества A , R , R' можно рассматривать как зависящие от вызова, т.е. в шкале статусов SSA формы, в зависимости от номера рассматриваемого вызова текущей процедуры берется либо первый статус (для первого варианта), либо n для n-го варианта, и проверяется , если он **u** , то это означает, что в этом варианте описания данной **def** переменной не было, и значит ее не нужно рассматривать в данном контексте вызова , если же статус **d** или **p** , что означает, что переменная либо обязательно определена, либо возможно определена, то тогда она включается в контекст данного вызова.

Множество аргументов оператора строится в процессе обхода оператора исходя из информации потокового процессора об обходе всех выражений и объектных выражений, т.е. фактически аргументы оператора вычислять не нужно, эту работу делает потоковый процессор ,но только за исключением вызовов процедур, где не все фактические параметры процедуры могут быть ее реальными аргументами (а потоковый процессор выдает информацию о всех аргументах процедуры), т.е. использоваться внутри ее тела, но это замечание касается только тех процедур , которые имеют имплементированные тела в анализируемой программе.

Таким образом, в построенной линейной модели программы мы знаем аргументы, результаты и сильные результаты всех простых операторов, а для структурных операторов мы знаем только их R, R ' множества, полученные из разницы контекстов после и до оператора и множество A - его аргументов, которое содержит только аргументы заголовка структурного оператора. Между тем аргументами структурного оператора является множество :

Если у данного структурного оператора N ветвей, то

A = < Аргументы заголовка >+< Аргументы 1 ветви > + ... + < Аргументы N ветви >,

< Аргументы ветви > = < Аргументы линейного участка >,

(Линейный участок: линейная последовательность операторов S1....Sn)

< Аргументы линейного участка > = < Начальный контекст > пересечь с <Все аргументы линейного участка >,

< Все аргументы линейного участка > = < Аргументы оператора S1 > + ... + < Аргументы оператора Sn >.

Поэтому при завершении обхода каждой ветви любого из структурных операторов, происходит вычисление множества аргументов данной ветви и добавление его в множество аргументов оператора, которому принадлежит данная ветвь.

Таким образом, когда для данной процедуры построена ее линейная модель , то мы уже имеем построенные множества A , R для всех ее операторов, и далее проводится рекурсивный обход по всем ветвям по всем вариантам вызовов данной процедуры, начиная с основной ветви - ветви текущей

процедуры и далее по всем ветвям ее структурных операторов, при этом когда обходится ветвь множество обязательных результатов для каждого оператора строится заново по множеству его результатов, исходя из номера варианта процедуры, и шкал статусов переменных . Нужно отметить, что любая ветвь как раз и является линейной моделью (реализует линейную последовательность операторов) программы.

Затем в соответствии с определением зависимости операторов происходит анализ зависимостей операторов данного линейного участка процедуры, и установленные зависимости между операторами передаются в модуль, отвечающий за анализ добротности информационных потоков, при этом множество аргументов, тоже рассматривается с учетом номера варианта процедуры .

Разработка этого модуля анализа добротности информационных потоков являлась второй подзадачей при разработке построенного анализатора и сейчас ее реализация будет описана подробнее .

4.2. Анализ добротности информационных потоков.

Итак, вторая подзадача. Суть ее заключается в том, чтобы для конкретного линейного участка программы с выявленными информационными связями, т.е. зависимостями между операторами, определить добротность его информационных потоков.

Реализация этой задачи вылилась в два модуля **InformationGraph.mod -** язык **Modula-2, Regular.ob2 -** язык **Oberon-2** , первый из которых содержит в себе весь набор операций для работы с информационными потоками , начиная от их вычисления и заканчивая операциями , которые использовались в формулировке критерия регулярности информационных потоков (вычисление пересечений потоков , получение усеченной реализации и т.д.) , а второй в чистом виде реализует проверку критерия регулярности информационных потоков, используя процедуры модуля **InformationGraph.mod.**

Основой модуля **InformationGraph.mod** является динамическая верхне-треугольная булевская матрица, которая содержит информационные связи текущей линейной схемы операторов. В матрице **i** -му столбцу (строке) соответствует **i** оператор (оператор с номером i) в рассматриваемой линейной схеме, где нумерация идет по порядку начиная с нуля. Таким образом, в матрице на пересечении **i** строки и **j** столбца стоит единица, если оператор **Si**

зависит от оператора **Sj** в смысле определения, данного в пункте **2. Постановка задачи**, в противном случае там стоит нуль.

В модуле реализовано множество различных оптимизаций работы с информационными потоками. Например, существует список уже вычисленных информационных потоков и их пересечений который пополняется при вычислении новых. Эта необходимая оптимизация связана с тем , что одни и те же информационные потоки и их пересечения требуется вычислять повторно. И поэтому в текущей реализации при необходимости использовать какой-то информационный поток (пересечение потоков) сначала просматривается набор уже вычисленных, и если обнаружен нужный , то он и используется , в случае же его отсутствия поток (пересечение потоков) вычисляется и сохраняется .

Сами потоки реализованы в виде динамических списков структур , каждый элемент которого содержит номер оператора , который ему соответствует . При этом элементы списка упорядочены , что позволяет оптимизировать работу с ними. Например, при поиске пересечения двух информационных потоков мы имеем возможность оборвать проверку вхождения оператора с данным номером в данный информационный поток, как только текущий номер оператора проверяемого информационного потока становится больше номера искомого оператора.

Также существует оптимизация связанная с хранением указателя на конец списка потока, что позволяет в том же поиске пересечения информационных потоков сразу проверить входит ли искомый оператор в интервал номеров операторов данного потока или нет, что может существенно упростить поиск.

"Реализация информационного потока " представлена в виде списка из двух элементов - номеров начального оператора и конечного оператора реализации информационного потока.

Интерфейс взаимодействия с модулем **InformationGraph.mod** представлен набором следующих процедур :

1) INTERSECT_REALIZATION_stream_AND_INFORMATION_stream(k , p) ;

Процедура , возвращающая пересечение реализации информационного потока T(Sk) с информационным потоком I(Sp) в виде списка номеров операторов .

2) IF_INTERSECTING_streams(k , p);

Возвращает True , если два информационных потока I(Sp) и I(Sk) являются пересекающимися , в противном случае возвращает False.

3) TRUNCATED_REALIZATION_stream(k,p);

Возвращает усеченную реализацию информационных потоков T(Sk:Sp) , так же представленную в виде упорядоченного списка номеров операторов.

4) INTERSECT_REALIZATION_STREAM_AND_INFORMATION_STREAM(Sk,Sp)

Процедура , возвращающая пересечение реализации информационного потока T(Sk) с информационным потоком I(Sp) в виде списка номеров операторов . Отличается от INTERSECT_REALIZATION_stream_AND_INFORMATION_stream только тем , что параметрами ее являются уже построенные информационные потоки .

5) INTERSECT_SOME_PSEUDO_STREAMS(I , Sp);

Эта процедура возвращает пересечение "псевдо" потоков с информационными потоками. Необходима, для нахождения общих элементов в некотором пересечении информационных потоков и каким то третьим информационным потоком .

6) COMPARE_STREAMS(Sk , Sp);

Возвращает True , если два информационных потока I(Sp) и I(Sk) являются эквивалентными , т.е. состоят из одних и тех же операторов , в противном случае возвращает False.

7) IF_INDEPENDENT_streams(k , p) ;

Возвращает True , если два информационных потока I(Sp) и I(Sk) являются независимыми , в противном случае возвращает False.

8) Information_Stream(k);

Возвращает информационный поток I(Sk);

9) Init(NUMBER_OF_OPERATORS);

Создается динамическая верхнетреугольная матрица инцидентности. Здесь NUMBER_OF_OPERATORS - количество операторов в рассматриваемой линейной модели .

10) AddLink(Dependent_Operator,Depends_From_Operator);

Добавляется новая связь между операторами . Dependent_Operator - номер зависимого оператора , Depends_From_Operator - номер оператора, от которого зависит оператор с номером Dependent_Operator .

Следует отметить , что реализация информационных потоков , их пересечений . реализаций , усеченных реализаций и т.д. полностью инкапсулировано в модуле **InformationGraph.mod.** Это дает возможность независимой оптимизации операций работы с ними. Как уже упоминалось ранее , были реализованы различные оптимизации , для увеличения скорости работы анализатора .

Модуль **Regular.ob2** , пользуясь приведенным выше интерфейсом , последовательно проверяет оба пункта критерия регулярности информационных потоков .

В соответствии с первым пунктом пробегаются все пары информационных потоков , если два потока оказываются независимыми , что проверяется с помощью процедуры IF_INDEPENDENT_streams (k , p) , то тогда ищется их пересечение, с помощью вызова функции INTERSECT_REALIZATION_stream_AND_INFORMATION_stream, далее, если оказывается, что это пересечение не пусто , то в соответствии с критерием информационные потоки не добротны .

Второй пункт критерия проверяется так же просто. Снова пробегаются все пары информационных потоков , если два потока оказываются пересекающимися , что проверяется с помощью процедуры IF_INTERSECTING_streams (k , p) , то процедурой TRUNCATED_REALIZATION_stream(k , p) вычисляется их усеченная реализация. Далее процедурой INTERSECT_REALIZATION_STREAM_AND_INFORMATION_STREAM пересекаем полученную усеченную реализацию с информационным потоком I(Sp). Далее процедурой INTERSECT_SOME_PSEUDO_STREAMS пересекаем результат предыдущей операции с потоком Information_Stream(k) и если оказывается, что он равен результату предыдущего пересечения (INTERSECT_REALIZATION_STREAM_AND_INFORMATION_STREAM) , т.е. псевдо потоки одинаковы (что проверяется с помощью процедуры

COMPARE_STREAMS), то тогда информационные потоки так же не являются добротными .

Модуль **Regular.ob2** также имеет интерфейс взаимодействия .
Который состоит из процедур:

1) Init(NUMBER_OF_OPERATORS);

2) AddLink(Dependent_Operator,Depends_From_Operator);

Которые "наследуются" от модуля **InformationGraph.def** и были описаны ранее. А также еще имеется процедура

3) REGULAR_STREAMS

, которая и проводит анализ заданной первыми двумя процедурами матрицы инцидентности на предмет добротности информационных потоков.

4.3. Взаимодействие подзадач.

Итак, как же в целом происходит работа построенного анализатора добротности информационных потоков? Во-первых, основой является потоковый процессор, благодаря обходу реализованному в модуле **Pass.ob2** , модулем **analyser.ob2** строится и аттрибутируется дерево, содержащее выделенные линейные участки анализируемой процедуры (т.е. модель данной процедуры, содержащая все ее линейные участки операторов). Далее после окончания прохода процедуры модуль **analyser.ob2** производит рекурсивный обход по построенному дереву, где для каждой линейной схемы для каждого варианта вызова происходит анализ информационных связей между операторами, входящими в эту схему, и вызовами процедур модуля **Pegular.ob2** модуль **InformationGraph.mod** создает новую матрицу инцидентности (информационных связей) для данного линейного участка и для данного варианта .

Как уже упоминалось в линейной схеме, операторы пронумерованы по порядку, и соответственно здесь каждому оператору текущей линейной схемы присваивается номер (который соответствует его порядковому номеру от начала линейного участка). И при проверке добротности информационных потоков используются только эти номера.

Далее после анализа и задания всех информационных связей происходит вызов функции модуля **Pegular.ob2 ,** которая вызовами процедур модуля **InformationGraph.mod** производит анализ всех информационных потоков данного линейного участка и в случае не добротности выводится подробная информация о том, какие потоки и почему не являются добротными. В случае недобротности информационных потоков выводится подробное сообщение о том, какие потоки данного варианта линейного участка программы не добротны и почему .

После обхода процедуры, вычисленные для нее множества A,R сохраняются в списке уже пройденных процедур, для того чтобы в дальнейшем эту информацию можно было использовать в местах вызова этих процедур. После этого начинается обход следующей процедуры.

5. Заключение.

Данная работа является попыткой создания нового программного процессора обеспечивающего анализ добротности информационных потоков в Оберон-программах, базирующегося на контекстно-чувствительном потоковом анализе программ с аппроксимацией обязательных информационных связей , который обеспечивается потоковым процессором . Построенный анализатор позволяет проводить как контекстно-чувствительный, так и контекстно не чувствительный анализ добротности информационных потоков , что является выходом за изначально планируемые рамки задачи .

Сам анализатор представляет собой две независимые части, одна из которых обеспечивает взаимодействие с потоковым процессором, а вторая, реализованная в виде независимого модуля, имея определенный интерфейс, отвечает за тестирование добротности информационных потоков, это дает возможность легко подключать анализатор к другим программным процессорам.

Необходимость создания подобного анализатора объясняется тем, что реализованные в нем критерии являются очень существенными показателями с точки зрения добротности программного продукта.

В настоящее время выдаваемые анализатором сообщения имеют достаточно сложную структуру . Поэтому в дальнейшем планируется создание потокового визуализатора , который позволит доступно визуализовать информационные потоки , и отмечать их недобротность .

В будущем данная работа может иметь несколько возможных продолжений, например, таких, как развитие ее в процессор , позволяющий исправлять нарушения добротности информационных потоков или построение процессора, разбивающего исходную программу на независимые информационные потоки вычисления данных , что может использоваться при распараллеливании исполнения программы на нескольких процессорах .

6. Список литературы.

[1] Поттосин И.В. , Добротность программ и информационных потоков , журнал "Открытые системы " № 6 1998 год , стр. 41-45 .

[2] Куксенко С.В. , Шелехов В.И. , Статический анализатор семантических ошибок периода исполнения , журнал "Программирование " .