Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

И. С. Запреев

АНАЛИЗ ДОБРОТНОСТИ ИНФОРМАЦИОННЫХ ПОТОКОВ В ПРОГРАММАХ НА ЯЗЫКАХ МОДУЛА-2/ОБЕРОН-2

Препринт
111

Новосибирск 2004

Данная работа описывает статический анализ добротности Модула-2/Оберон-2 программ. В рамках работы реализованы два критерия добротности информационных потоков, а именно, критерии регулярности и подтвержденности.

Критерии добротности информационных потоков реализованы на базе потокового анализа. Информация, получаемая в процессе работы анализатора, сохраняется в XML-формате. Реализован графический интерфейс, позволяющий доступно визуализировать информационные потоки и их недобротности, а также предоставлять дополнительную потоковую информацию.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

I. V. Zapreev

ANALYSIS OF INFORMATION FLOW QUALITY IN MODULA-2/OBERON-2 PROGRAMS

Preprint
111

Novosibirsk 2004

This paper describes the static analysis of quality of Modula-2/Oberon-2 programs. It presents implementation of two criteria of information flow quality, namely, the criteria for being regular and confirmed.

The criteria of information flow quality are implemented on the basis of data flow analysis. The information derived by analyzer is stored in the XML-format. We have implemented a graphical interface that allows us to visualize information flows and the locations of their low-quality, as well as to provide additional flow information.

1. ВВЕДЕНИЕ

Для оценки внутренних достоинств реализации программы с технической стороны в работе [8] было предложено понятие добротности программы. В термин добротность вкладывается такой аспект хорошей программы, который заключается в том, что программа разумно, рационально реализована, с достаточно продуманной организацией потоков управления и информационных потоков, не переусложнена. Позднее критерии разумной организации информационных потоков были точно сформулированы в работе [1].

Содержательно эти критерии могут быть выражены следующим образом:

- критерий регулярности информационных потоков заключается в проверке отсутствия неоправданной перепутанности информационных потоков;
- критерий подтвержденности потоков заключается в проверке отсутствия использования неинициализированных переменных.

Разумная организация информационных потоков в программе повышает ее надежность, при этом надежность рассматривается как один из критериев правильности программы. В этом смысле, критерии не заключаются в проверке соответствия программы ее спецификациям, но помогают определить возможные ошибки в программе.

Данная работа описывает реализацию критериев добротности информационных потоков для программ, написанных на языках Модула-2/Оберон-2, и может быть разбита на несколько частей, а именно:

- разработка алгоритмов проверки критериев добротности, сформулированных в работе [1] на основе данных о программе, полученных в результате потокового анализа;
- реализация критериев регулярности и подтвержденности информационных потоков;
- реализация многофункционального графического интерфейса, который позволяет не только доступно визуализировать обнаруженные нарушения критериев, далее недобротности, но и предоставляет широкие возможности по визуализации данных, полученных в результате потокового анализа программы.

Разработанный анализатор проводит статический анализ программы в соответствии с критериями добротности информационных потоков для

программ на языках Модула-2/Оберон-2 на основе данных, полученных в результате потокового анализа программы.

Исходя из содержательной формулировки критериев, стоит отметить, что критерий регулярности не выявляет семантические ошибки в программе, а лишь обнаруживает некоторую запутанность вычислений, которая может являться следствием других ошибок или в дальнейшем стать их причиной. Данный критерий позволяет выявить цепочки операторов на линейном участке программы, которые могут быть выполнены параллельно. Критерий подтвержденности в свою очередь позволяет отследить не только использование неинициализированной переменной, что может являться ошибкой, но также помогает определить вспомогательную информацию. Например, для условного оператора критерий определяет, на какой из его ветвей не было инициализации переменной.

Графический интерфейс системы в первую очередь призван визуализировать нарушения описанных выше критериев добротности, но так как критерии основываются на информационных связях в программе, то наряду с нарушениями визуализируются информационные потоки, а также иная вспомогательная информации о программе.

В разд. 2 предоставлена дополнительная информация о критериях добротности, а также приведены точные формулировки критериев регулярности и подтвержденности. Разд. 3 содержит описание потокового процессора, который предоставляет потоковую информацию о программе. Разд. 4 описывает функциональные возможности реализованной системы. Разд. 5 дает общее представление о реализации системы. Разд. 6 и 7 описывают реализацию критериев регулярности и подтвержденности соответственно. Разд. 8 описывает способ сохранения результатов анализа. В разд. 9 дается обзор работ по данной тематике, а разд. 10 является заключением данной работы, в котором также упомянуто об ее дальнейшем развитии.

2. КРИТЕРИИ ДОБРОТНОСТИ

Остановимся на следующей формулировке добротности программ, которая дается в статье [1]. Добротность в ней определяется, как «разумная, рациональная, не переусложненная организация информационных потоков и потоков управления, а также разумное построение вычислений». Суженное таким образом понятие добротности программы не касается таких критериев, как, например, соответствие её спецификации.

Исходя из данного содержательного определения добротности, формулируются следующие группы критериев: количественные, генетические, структурные и прагматические.

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). Наиболее полный обзор таких мер сложности, существующих в настоящем времени, содержится в статье [3].

Генетические критерии основаны на доверии некоторым заведомо хорошим технологиям. 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 — разумная организация потока управления предполагает синтаксически его ясное отображение в программном тексте. Подход к такому отображению сформулирован давно и выражается понятием структурированной программы (программы без операторов goto). В структурированной программе управление представлено в виде иерархии регулярных управляющих структур, каждая из которых имеет один вход и один выход. Помимо такого простого требования, исключающего хаотическое переплетение потоков управления, для добротной программы с регулярностью управления важно иметь хорошо продуманную процедурную основу.

Прагматические критерии связаны с тем, что формально можно выявить некоторые свойства программы, которые содержательно можно трактовать как цели, достижению которых служит программа. Такой целью может быть конечное состояние переменных, порождаемое программой, множество ее результатов, потоки управления, информационные потоки и т.п. Развитые методы анализа программ могут обнаруживать подобные цели и находить те фрагменты и объекты данной программы, которые заведомо никак не служат достижению такой формально выявляемой цели. Наличие таких излишеств и служит основанием признать программу недобротной в соответствии с прагматическими критериями.

Класс прагматических критериев описан в статье [8]. Работа [4] дает точное определение ряда «излишеств», которые являются нарушениями

этих критериев, и определяет каноническую форму, автоматически исключающую данные несообразности.

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

Множество прагматических критериев группируется по нескольким классам[1].

1. Целевая направленность. Для этого вида критериев под целью понимается множество допустимых конечных значений переменных. Вычислительный процесс за счет применения операторов программы получает последовательность состояний — от начального до конечного, а программа, описывающая множество таких процессов, не может содержать операторы, всегда сохраняющие исходное (для оператора) состояние, в противном случае мы не движемся к цели, а топчемся на месте.

2. Структурная целесообразность. Целью здесь является поддержание видимой структуры управления, а за нецелесообразность признается наличие циклов, заведомо не выполняемых более одного раза, или таких, эффект которых не требует повторения, а также операторов ветвления с заведомо невыполняемыми ветвями.

3. Оправданная выстроенность вычислений. Целью здесь является некоторое разумное размещение вычислительного оператора в вычислительном процессе, т.е. там, где его присутствие необходимо.

4. Вычислительная неизбыточность. Целью здесь является выработка значений тех переменных, которые формально могут быть определены как результаты программы, а излишеством считается присутствие вычислений и объектов, никак не влияющих на эти результаты.

5. Разумная организация информационных потоков. Здесь целью является поддержание обнаруженных информационных потоков, а отклонением от цели — «дурная» (хотя и правильная) организация таких потоков. В отличие от предыдущих видов требований, которые сформулированы точно, в [8] дается интуитивное представление о том, что следует считать хорошей организованностью. В статье [1] дается точное определение, которое и будет приведено ниже.

Добротность информационных потоков разделяется на **регулярность** (отсутствие неоправданной перепутанности информационных потоков) и **подтвержденность** (что означает, что любая переменная всегда будет инициализирована перед своим использованием). Более точные формулировки обоих критериев будут приведены далее.

В данной работе описывается анализатор добротности информационных потоков в программах на языках Модула-2/Оберон-2, в котором реали-

зована проверка критериев **регулярности и подтвержденности.** Данный анализатор также обеспечивает визуализацию обнаруженных недобротностей.

В соответствии со статьей [1] , ниже будет приведено точное определение критериев добротности информационных потоков; но прежде рассмотрим несколько определений и модель анализируемой программы, в контексте которых в работе [1] и даются данные критерии.

Стоит отметить, что работа И. В. Поттосина [1], содержащая точные формулировки упомянутых прагматических критериев, базируется на работе S. Pan, R.G.Dromey. Bejond Structured Programming [4] и в этом смысле развивает заложенные в ней идеи добротности информационных потоков.

Для определения добротности информационных потоков используется модель программы в виде линейных схем, в которой программа представляется в виде линейной последовательности операторов $S_1, S_2, \ldots, S_n$, и для каждого S_i известны множества: $A(S_i)$ — множество его аргументов, — множество результатов и $R'(S_i)$ – множество обязательных результатов. При этом $R'(S_i)$ — подмножество $R(S_i)$, содержащее только те результаты, которые обязательно будут выработаны при любом выполнении оператора S_i .

Рассмотрим линейную схему $T = S_1 \ldots S_n$.

По определению, говорят, что S_j *зависит от* S_i (i<j) , если:

- существует такая переменная x, что x принадлежит $R(S_i)$, x принадлежит $A(S_j)$, x не принадлежит $R'(S_{i+1} \ldots S_{j-1})$. Это означает, что значение x выработанное Si доходит до Sj , где потом используется;
- существует такая переменная x, что x принадлежит $R(S_i)$, x принадлежит $R(S_j) \setminus R'(S_j)$, x не принадлежит $R'(S_{i+1} \ldots S_{j-1})$. Иными словами, значение, выработанное S_i , доходит до S_j , а тот его меняет или нет. При этом если значение не меняется, то тогда его можно считать псевдоаргументом S_j .

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

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

Реализацией $I(S_k)$ называется подпоследовательность

$$T(S_k) = S_p, \ldots, S_k ,$$

где p — это минимальный номер вершины в $I(S_k)$.

Информационные потоки $I(S_k)$ и $I(S_j)$ 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 $I(S_n)$ по отношению к пересекающемуся с ним $I(S_m)$ называется такая последовательность операторов $T(S_n : S_m)$, для которой оператор с минимальным в подпоследовательности номером берется без учета номеров общих для данных информационных потоков операторов.

Основываясь на описанных выше определениях, в работе [1] вводится следующая точная формулировка критерия добротности информационных потоков с точки зрения их регулярности.

Критерий регулярности

Информационные потоки в программе добротны с точки зрения регулярности, 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 $I(S_n)$ *по отношению к пересекающемуся с ним* $I(S_m)$ *не содержит операторов, для которых соответствующие вершины принадлежат* $I(S_m)$ *и не принадлежат* $I(S_n)$.

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.

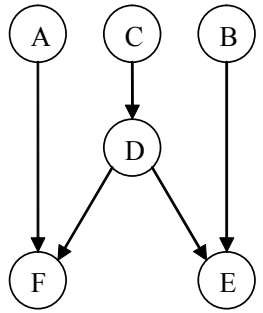
Пример:

Пусть задана 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 $I(F)$. Поэтому последовательность ABCDEF недобротна с точки зрения критерия регулярности, в тоже время видно, что последовательность вычислений CDAFBE является добротной.

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.

Для каждого оператора S программы можно построить линейную схему $\overline{S}$ операторов, предшествующих S в простом (без повторений) пути от начала программы, и линейную схему *S* операторов, следующих за S в простом пути до завершения программы. Структурные операторы раскрываются в этих линейных схемах, только если S содержится в них.

В работе [1] также приводится следующая формулировка критерия подтвержденности информационных потоков.

Критерий подтвержденности

1. *Если существует переменная x такая, что* $x \in A(S)$, *то* $x \in R(\overline{S})$.
2. *Если оператор* S *есть оператор ветвления с ветвями* $SL_1,\ldots,SL_n$ *и переменная x такова, что* $x \notin R(\overline{S})$, *то должно быть справедливо хотя бы одно из следующих условий:*
   a) $x \notin \bigcup R(SL_i)$;
   b) $x \in \bigcap R(SL_i)$;
   c) $x \in R(SL_i)$ *и* $x \notin A(\underline{S})$.
3. *Если оператор* S *есть оператор цикла с телом* SL *и* $x \notin R(\overline{S})$, *то должно быть справедливо хотя бы одно из условий:*
   a) $x \notin R(SL)$;
   b) $x \in R(SL)$ *и гарантируется, что при первом исполнении* S *он исполняется как минимум один раз;*
   c) $x \in R(SL)$ *и* $x \notin A(\underline{S})$.

Примером нарушения критерия подтвержденности может служить следующий текст программы:

```
PROCEDURE FUNC(VAR SMTH:BOOLEAN):INTEGER;
 VAR
A:INTEGER;
 BEGIN
IF (SMTH) THEN
A := 1;
 END;
 RETURN A;
 END FUNC;
```

В данной процедуре в зависимости от значения ее параметра переменная A может быть не инициализирована, что приведет к использованию переменной с неопределенным значением в операторе RETURN.

Критерий подтвержденности информационных потоков несколько ограничивает свободу программиста, но тем не менее, соответствует хорошей дисциплине и культуре программирования.

3. ПОТОКОВЫЙ ПРОЦЕССОР

Критерий добротности информационных потоков предполагает наличие некоторых знаний о самой программе. Так, для каждого оператора S_i необходимо знать множество его аргументов $A(S_i)$, множество его результатов $R(S_i)$ и множество его сильных результатов $R'(S_i)$. Таким образом, анализатор добротности информационных потоков естественно рассматривать как «языковой процессор», работающий после другого программного процессора — потокового процессора, который реализует потоковый анализ [5].

Потоковый процессор является частью статического анализатора WASP (Модула-2/Оберон-2 Static Analyzer) для программ на смеси языков Модула-2/Оберон-2 в окружении системы программирования XDS для Linux и Windows 95/NT. Принципы работы статического анализатора описаны в статье [2].

В системе WASP потоковый процессор является предпроцессором статического анализатора семантических ошибок периода исполнения. Описываемый в данной работе анализатор также использует потоковый процессор в качестве предпроцессора.

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, содержащий интерфейс обхода программы и доступа к потоковой информации. На основе данных, полученных с помощью этого модуля, строится дерево линейного представления программы, необходимое анализатору добротности информационных потоков.

Потоковый процессор реализует межпроцедурный контекстно-чувствительный анализ, с аппроксимацией как обязательных, так и возможных информационных связей [6]. Контекстно-чувствительный анализ сохраняет специфику каждого отдельного вызова процендуры при интеграции потоковой информации для разных вызовов процедур.

Абстрактные — это переменные, принадлежащие потоковой модели программы, реализуемой потоковым процессором.

А любая переменная памяти аппроксимируется единственной абстрактной переменной.

Абстрактные переменные классифицируются следующим образом:

- Основные переменные — соответствующие переменным, описанным в программе.
- New переменные — генерируемые при исполнении операторов ALLOCATE, NEW.
- Компонентные переменные — компоненты структурных переменных типов ARRAY, RECORD.

Для представления информационных связей в потоковом процессоре применяются SSA [7] формы (static single assignment form), которые позволяют связывать точку использования переменной с точкой ее определения, что обеспечивает для каждого использующего вхождения переменной одно определяющее.

На месте каждого определяющего вхождения переменной X в программе 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", ”i” — 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 fi-функции, которые служат для определения значений переменных, при слиянии различных ветвей исполнения программы, например, после оператора if.

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

Контекстно-чувствительный анализ для каждой процедуры характеризуется тем, что в общем случае реализуется несколько вариантов анализа, соответствующих различным ее вызовам. Различные вызовы процедуры фиксируются набором вариантов процедуры. Если число вариантов больше одного, то для всех определений переменных процедуры, а также для всех заместителей вместо одного <статуса> используется <вектор статусов>.

Итак, если есть использующее вхождение абстрактной 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.

В целом работа потокового процессора заключается в построении некоторого древовидного представления программы, дальнейшего обхода и вычисления потоковой информации. 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 по-процедурно. Обход происходит по всем операторам тела процедуры.

4. ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ СТАТИЧЕСКОГО АНАЛИЗАТОРА НЕДОБРОТНОСТЕЙ

Прежде чем приступить к детальному описанию реализации разработанной системы, в этом разделе будет представлено описание реализованной функциональности системы. В системе были реализованы следующие возможности.

1. Анализ добротности информационных потоков программах, написанных на языках Модула-2/Оберон-2 в соответствии с критериями регулярности и подтвержденности. Система строит дерево программы, в котором каждый оператор атрибутирован множествами

аргументов и результатов, а каждая ветвь программы атрибутирована начальным и конечным контекстами. Реализуется проверка критериев регулярности и подтвержденности на основе построенного дерева программы и потоковой информации, предоставляемой потоковым процессором.

2. Сохранение дерева программы и результатов анализа в XML-документ [9]. Соответствующая DTD-грамматика [9] приведена в приложении 1. При сохранении дерева программы также сохраняются контексты ветвей, аргументы и результаты операторов. Обнаруженные нарушения критериев регулярности и подтвержденности сохраняются в вершинах XML-документа, представляющих соответствующие линейные участки программы.
3. Визуализация информации о дереве программы, недобротностях информационных потоков, зависимостях между операторами и т.д. (см. приложение 2).

Графический интерфейс (приложение 2, рис. 1*)* является независимой компонентой системы и служит для визуализации результатов анализа. На вход визуализатор получает XML-документ, построенный анализатором добротностей информационных потоков, и предоставляет пользователю следующие возможности:

1. *Визуализация дерева программы.* Пользователю доступно дерево программы для навигации по нему (приложение 2, рис. 2*)*, а также для получения дополнительных данных об операторах программы и ее линейных участках. А именно, аргументов и результатов операторов (приложение 2, рис. 10*)*, начальных и конечных контекстах линейных участков (приложение 2, рис. 9*)*, а также зависимостях между операторами на линейных участках программы (приложение 2, рис. 8*)*.
2. *Возможность навигации по дереву программы с подсветкой соответствующих строк входного текста программы.* Во время навигации по дереву открываются соответствующие файлы входного текста программы. А при выборе оператора происходит его подсветка во входном тексте программы (приложение 2, рис. 2*)*.
3. *Многооконный интерфейс.* Возможностью иметь одновременно несколько открытых файлов входного текста программы приложение 2, рис. 2*)*.
4. *Адекватное отображение различного рода операторов в дереве программы.* В дереве программы различным образом отображены модули, процедуры, операторы, комплексные операторы, ветви

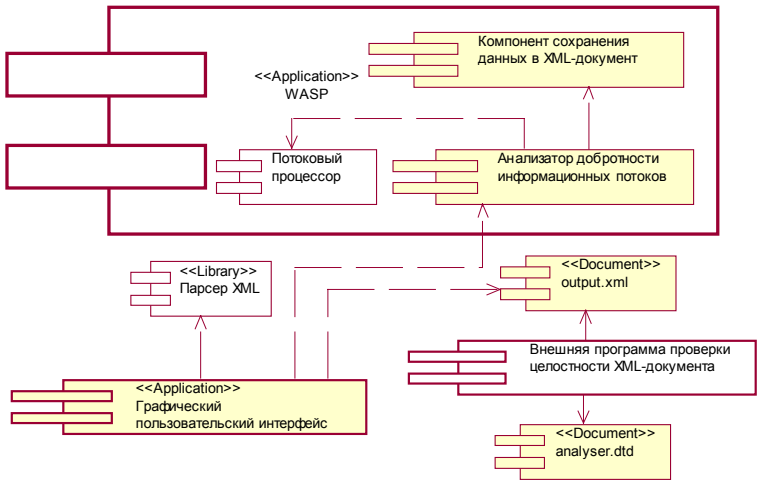
процедур и комплексных операторов. Также отдельно выделены операторы вызова и визуализированы все вызовы процедур, происходившие из выражений — аргументов данного оператора.

5. *Визуализация полной информации об обнаруженных недобротностях информационных потоков.* Имеются два независимых списка ошибок регулярности и подтвержденности информационных потоков (приложение 2, рис. 3–7, 11*)*.
6. *Возможность получения дополнительной информации о программе.* Через контекстное меню в дереве программы имеется возможность получить следующую дополнительную информацию:
   a) множество аргументов и результатов любого выбранного оператора (приложение 2, рис. 10*)*;
   b) визуализация начальных и конечных контекстов линейного участка программы (приложение 2, рис. 9*)*;
   c) визуализация зависимостей между операторами на выбранном линейном участке программы (приложение 2, рис. 8*)*.
7. *Возможность прямой и обратной локализации недобротностей в анализируемой программе*.
   a) *Прямая локализация.* Вершины дерева программы, такие как модули, процедуры, структурные операторы, которые содержат в своей структуре ветви линейных участков с найденными недобротностями, подсвечиваются красным цветом. Таким образом, по дереву сразу видны участки программы, содержащие недобротности. При выделении такой вершины в списках ошибок происходит подсветка всех недобротностей, лежащих на вложенных в данную вершину линейных участках программы.
   b) *Обратная локализация.* При выборе недобротности в одном из списков (подтвержденность/регулярность) происходит не только показ информации о недобротности во входном тексте программы, но также происходит локализация соответствующего линейного участка программы в ее дереве.
8. *Предоставляется расширенная информация о недобротностях.* Сообщение о недобротности информационных потоков как регулярности, так и подтвержденности, разбито на подстроки, обозначенные разными цветами (приложение 2, рис. 3, 11). Данные подстроки выделяют множества операторов и имеют соответственные обозначения. При выделении предупреждения о недобротности, в тексте программы подсвечиваются все операторы, относящиеся к

данной недобротности. А при выделении подстроки в сообщения, подсвечиваются только соответствующие множества операторов (приложение 2, рис. 3-7).

5. РЕАЛИЗАЦИЯ СИСТЕМЫ

С точки зрения верхнеуровневой реализации система разбита на следующие компоненты:



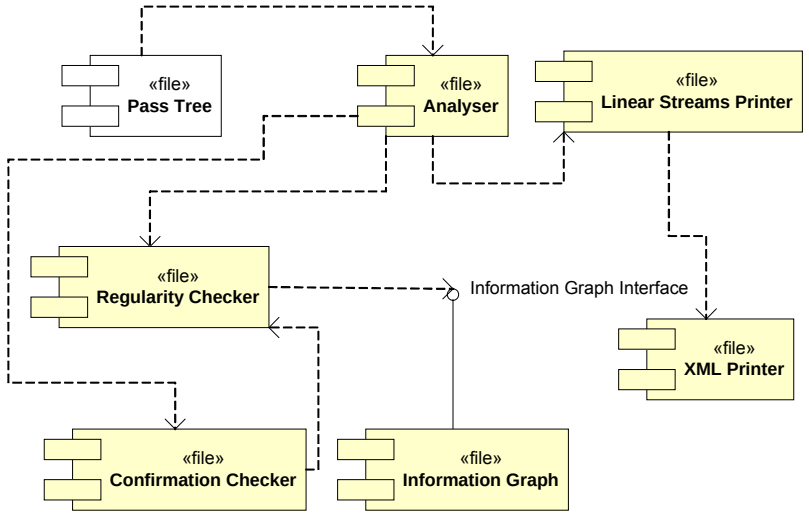
На приведенной выше диаграмме компонентов цветом выделены те части системы, которые были реализованы в рамках данной работы.

Способы реализации компонентов системы:

1. *Графический пользовательский интерфейс* —
   Java, Swing, Apache Xerces 1.2.0
2. *output.XML, analyser.DTD* —
   XML 1.0.
3. *Анализатор добротности информационных потоков* —
   Модула-2/Оберон-2.
4. *Компонент сохранения данных в XML-документе* —
   Оберон-2.

5.1. Основные компоненты анализатора

Следующая диаграмма компонентов представляет детальное строение компонента анализатора добротностей информационных потоков и компонента сохранения результирующих данных в XML.



На приведенной выше диаграмме компонентов цветом выделены те части подсистемы, которые были реализованы в рамках данной работы.

Pass Tree это основной модуль обхода дерева программы статического анализатора WASP, который был незначительно модифицирован — добавлены вызовы процедур модуля **Analyzer**.

Задачей модуля **Analyzer** является построение дерева программы, он также координирует работу модулей анализаторов добротностей информационных потоков и модуля сохранения данных в формате XML.

Модули **Regular Checker** и **Confirmation Checker** занимаются анализом регулярности информационных потоков и подтвержденности информационных потоков соответственно.

Information Graph Interface представляет интерфейс модуля работы с информационными потоками, который используется модулем **Regular Checker**. Модуль **Information Graph** реализует данный интерфейс.

Модули **Linear Streams Printer** и **XML Printer** обеспечивают обход результирующего дерева программы при сохранении его в формате XML и также его печать в результирующий XML документ.

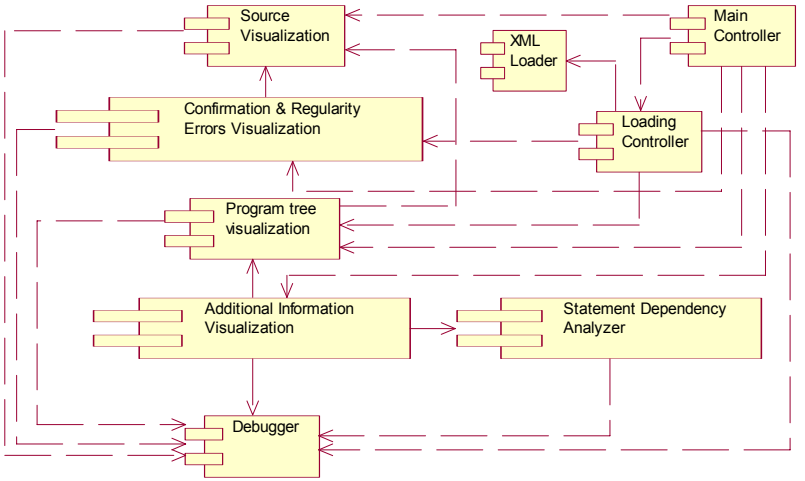
5.2. Реализация Графического интерфейса системы

Как уже было отмечено, графический интерфейс системы был написан на языке программирования Java (jdk 1.3) с использованием awt и swing, а также в качестве XML DOM парсера использовался пакет классов для работы с XML — Apache Xerces 1.2.0. Анализатор WASP может использоваться на Windows9X/Windows NT/Unix платформах, таким образом, использование java обеспечивает полную переносимость графического интерфейса.

В текущей реализации система разбита на следующие пакеты.

- Базовый пакет системы. Также содержит классы необходимые для запуска приложения, класс основного окна системы, класс контроллера загрузки XML-документа, класс, отвечающий за подсветку информационных потоков, недобротностей и операторов во входном тексте анализируемой программы. Пакет также содержит некоторые другие вспомогательные классы.
- Пакет, содержащий классы отладчика системы. Отладчик системы обеспечивает создание файла отладочной информации системы. Существуют несколько уровней отладки, любой класс графического интерфейса может регистрироваться в отладчике для вывода информации о своей работе в файл отладочной информации системы.
- Пакет, содержащий базовые классы ошибок, выявленных критериями регулярности и подтвержденности. Абстрактные классы и интерфейсы списков сообщений о выявленных недобротностях. Содержит классы, реализующие визуализацию дерева программы, и классы, отвечающие за загрузку XML-документа.
- Пакет, содержащий классы необходимые для реализации анализа зависимостей между операторами на линейных участках программы, а также для визуализации обнаруженных зависимостей.
- Пакет, содержащий классы, поддерживающие работы контекстного меню дерева программы.
- Пакет, содержащий классы реализации списка нарушений критерия подтвержденности и классы, отвечающие за его визуализацию, а также классы, представляющие ошибки подтвержденности.
- Пакет, содержащий классы, реализующие список нарушений критерия регулярности и классы, отвечающие за его визуализацию, а также классы, представляющие ошибки регулярности.

Графический интерфейс включает в себя следующие компоненты:

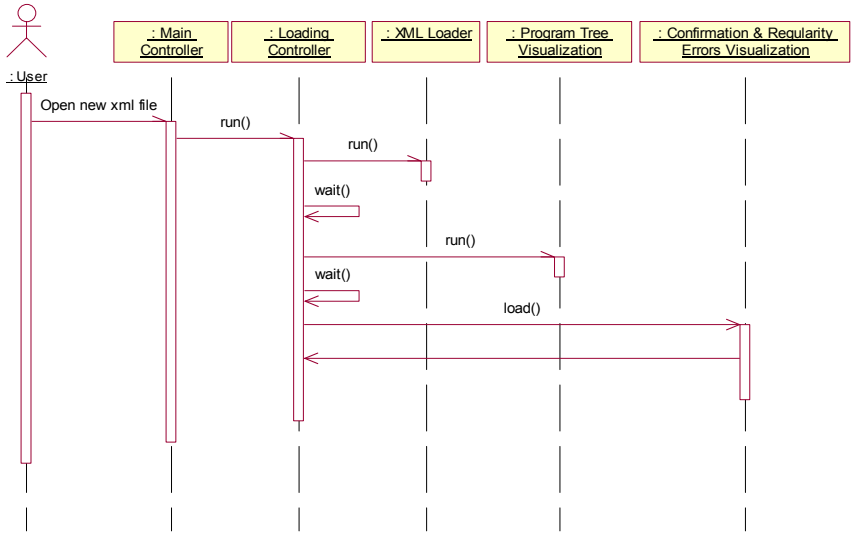


- **Main Controller** — Центральный компонент системы, отвечающий за координацию работы всего приложения:
  - первичную инициализацию приложения,
  - загрузку XML-документа,
  - координацию работы других компонент системы.
- **Loading Controller** — Компонент приложения, работающий в независимом от основного потоке управления. Обеспечивает загрузку XML-документа DOM парсером в независимом потоке управления. Также в независимом потоке производится инициализация дерева программы: компонента Program tree visualization. После этого инициализирует компонент Confirmation & Regularity Errors Visualization.
- **Debugger —** Независимый компонент приложения, с помощью которого производится сбор информации о стабильности работы системы. Основными его задачами являются:
  - регистрация классов с определенным уровнем отладки,
  - вывод поступаемой от классов отладочной информации в соответствии с их текущем уровнем отладки.
- **XML Loader** — Отвечает за загрузку XML-документа.
- **Program tree visualization** — Компонент, отвечающий за визуализацию дерева программы.

- **Source Visualization** — Компонент, задачей которого является визуализация входного текста анализируемой программы, а также подсветка необходимой его части (операторов, информационных потоков и т.д.).
- **Confirmation & Regularity Errors Visualization —** Компонент визуализации списков нарушений критериев регулярности и подтвержденности.
- **Statement Dependency Analyzer** — Данный компонент обеспечивает анализ зависимостей операторов выбранного линейного участка программы.
- **Additional Information Visualization** — Обеспечивает визуализацию зависимостей между операторами на выбранном линейном участке программы, визуализацию начального и конечного контекстов выбранной ветви программы, визуализацию множества аргументов и результатов, выбранного в дереве программы оператора.

Реализация классов системы является стандартной, поэтому их описание приводиться не будет.

На приведенной ниже диаграмме последовательностей действий (sequence diagram) приведено описание загрузки XML-документа в системе графического интерфейса:



Вызовы методов run() означают запуск работы компонентов в отдельных потоках управления. Вызовы методов wait() означают, что компонент Loading Controller ожидает завершения запущенного потока исполнения, при этом поток управления, в котором находится Loading Controller, периодически засыпает и просыпается для того, чтобы проверить, завершено ли исполнение запущенного им дочернего потока управления. Методы run() реализованы таким образом, чтобы ни использовать всех ресурсов процессора, для этого используется периодический вызов из них оператора sleep.

В целом работа графического интерфейса подчиняется правилам работы графических оболочек, реализованных с использованием библиотеки swing.

6. ПРОВЕРКА КРИТЕРИЯ РЕГУЛЯРНОСТИ

В связи с тем, что потоковый процессор реализован на языках Модула-2/Оберон-2, а построенный анализатор должен тесно интегрироваться с текстом программы процессора, то для написания анализатора регулярности информационных потоков были выбраны языки Модула-2/ Оберон-2 .

При реализации вся задача по построению анализатора для данного критерия была разбита на две подзадачи.

1. **Построение линейной модели программы.** Это получение необходимой информации от потокового процессора об анализируемой программе, разбиение ее на линейные последовательности операторов, вычисление для всех операторов множеств их аргументов, результатов и обязательных результатов, а также определение информационных связей между операторами.
2. **Анализ добротности информационных потоков по построенной модели.**

6.1. Построение линейной модели программы

Как уже упоминалось, потоковый процессор сначала строит некоторое древовидное представление программы, и потом в дальнейшем обходит его, атрибутируя, вычисляя текущий контекст. Обход построенного и атрибутированного дерева осуществляется в модуле **Pass Tree**.

При обходе, реализуемом в модуле **Pass Tree**, вызовами процедур модуля **Analyzer** строится древовидная структура программы, которая реализует линейную модель программы. В строящейся структуре каждый оператор является либо простым оператором, либо составным. В последнем случае он содержит набор ветвей, которые представляют собой последова-

тельные списки операторов, содержащихся на этих ветвях. Для оператора **if** ветви две; 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.

Нужно отметить, что линейная модель программы реализована в виде динамических списков структур.

Ниже будет описано, как строятся множества A , R , R' для операторов линейной схемы программы.

При последовательном обходе потоковым процессором тела процедуры, при прохождении операторов запоминается текущий контекст переменных — контекст перед оператором, далее, после прохождения оператора, запомненный контекст сравнивается с контекстом после пройденного оператора, и из разницы контекстов определяется множество результатов R пройденного оператора. Также в процессе обхода оператора добавляется соответствующая ему структура в строящуюся линейную модель программы, и полученное множество результатов оператора R сохраняется в соответствующем поле добавленной структуры.

Так как контекст является множеством переменных, действующих в данной точке программы и имеющих соответствующие SSA-формы, то и множество результатов оператора R тоже является набором SSA-форм переменных.

На основе SSA-формы из множества R вычисляется множество R' — обязательных результатов оператора, при этом так как потоковый процессор проводит контекстно-чувствительный анализ программы, то SSA-форма содержит информацию обо всех вызовах анализируемой процедуры. Следовательно, 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 его значение. Если статус **u**, это означает, что в этом варианте описания данной **def**-переменной не было, и ее не нужно рассматривать в данном контексте вызова, если же статус **d** или **p**, то переменная либо обязательно определена, либо возможно определена, и тогда она включается в контекст данного вызова.

Информацию о множестве аргументов оператора предоставляет потоковый процессор, за исключением вызовов процедур, где не все фактические параметры процедуры могут быть ее реальными аргументами.

Таким образом, в построенной линейной модели программы легко определяются аргументы, результаты и сильные результаты всех простых операторов. Для структурных операторов известны только множества R, R' и множества аргументов A, которые изначально содержат только аргументы заголовков структурных операторов. Между тем, аргументами структурного оператора, имеющего в общем случае N ветвей, является множество:

$$A = < Аргументы\ \_ заголовка > \cup < Аргументы\ \_ 1 \_ ветки > \cup \ldots \cup < Аргументы\ \_ N \_ ветки >$$

,

$$< Аргументы \_ i \_ ветви > = < Аргументы \_ линейного \_ участка >,$$

Линейный участок — это линейная последовательность операторов $S_1 \ldots S_n$,

$$< Аргументы \_ линейного \_ участка > = < Начальный \_ контекст > \cap < Все \_ аргументы \_ линейного \_ участка >$$

$$< Все \_ аргументы \_ линейного \_ участка > = A(S_1) \cup \ldots \cup A(S_n)$$

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

После того как для процедуры построена линейная модель, она содержит построенные множества A, R для каждого из операторов. 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 обходе ветви множество обязательных результатов для каждого оператора строится заново, исходя из номера варианта вызова процедуры.

Для каждого из линейных участков производится анализ зависимостей операторов и проверка критерия регулярности информационных потоков, при этом множества аргументов операторов тоже рассматриваются с учетом номера варианта вызова процедуры.

Ниже приведено описание алгоритма проверки критерия регулярности.

6.2. Анализ регулярности информационных потоков

Суть задачи заключается в том, чтобы для конкретного линейного участка программы с выявленными информационными связями, т.е. зависимо-

стями между операторами, определить добротность его информационных потоков.

Реализация этой задачи свелась к созданию двух модулей **Information Graph** и **Regularity Checker.** Первый модуль содержит в себе весь набор операций для работы с информационными потоками, начиная от их вычисления и заканчивая операциями, которые использовались в формулировке критерия регулярности. Второй модуль в чистом виде реализует проверку критерия регулярности информационных потоков, используя процедуры модуля **Information Graph.**

Основой модуля **Information Graph** является динамическая верхнетреугольная булевская матрица, которая содержит информационные связи текущей линейной схемы операторов.

В матрице **i**-му столбцу (строке) соответствует **i**-й оператор (оператор с номером **i**) в рассматриваемой линейной схеме, где нумерация идет по порядку, начиная с нуля. Таким образом, в матрице на пересечении **i**-й строки и **j**-го столбца находится единица, если оператор S_i зависит от оператора S_j в смысле определения, данного в разд 2**.**, в противном случае в данной позиции находится 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. Поэтому в текущей реализации, при необходимости использования какого-либо информационного потока (пересечение потоков), сначала производится поиск по набору уже вычисленных информационных потоков.

Информационные 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 проверка вхождения оператора с данным номером в данный информационный поток завершается в случае, если 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 информа-

ционных потоков сразу проверить, входит ли искомый оператор в интервал номеров операторов данного потока или нет.

«Реализация информационного потока» представлена в виде списка из двух элементов — номеров начального оператора и конечного оператора реализации информационного потока.

Интерфейс взаимодействия с модулем **Information Graph** представлен набором процедур, которые реализуют все методы работы с информационными потоками, необходимые для проверки критерия регулярности, например, вычисление пересечения информационных потоков, усеченной реализации одного потока относительно другого и т.д.

Следует отметить, что реализация информационных потоков, их пересечений, реализаций, усеченных реализаций и т.д. полностью инкапсулирована в модуле **Information Graph.** Это дает возможность независимой оптимизации операций работы с ними.

Модуль **Regularity Checker** содержит реализацию проверки критерия регулярности, с использованием интерфейса **Information Graph** проверяются оба пункта критерия.

6.3. Взаимодействие подзадач

Итак, как же в целом происходит анализ критерия регулярности информационных потоков? Во-первых, основой является потоковый процессор, благодаря обходу, реализованному в модуле **Pass Tree**. Модулем **Analyzer** строится и аттрибутируется дерево, содержащее выделенные линейные участки анализируемой процедуры. Далее, модуль **Analyzer** 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. После, вызовами процедур модуля **Regularity Checker**модуль **Information Graph**создает новую матрицу инцидентности (информационных связей) для данного линейного участка и для данного варианта вызова процедуры.

Как уже упоминалось, в линейной схеме операторы пронумерованы по порядку, и соответственно, каждому оператору текущей линейной схемы присваивается номер. При непосредственной проверке критерия регулярности информационных потоков используются уже только номера операторов.

После анализа и задания всех информационных связей происходит активизация модуля **Regularity Checker ,** который с помощью процедур модуля **Information Graph** производит анализ всех информационных потоков

данного линейного участка. В случае обнаружения нарушений критерия регулярности подробная информация о каждом из нарушений сохраняется в объект, представляющий нарушение регулярности, который сохраняется на ветви дерева, соответствующей анализируемому линейному участку процедуры. В дальнейшем, после завершения анализа регулярности и подтвержденности всей программы, данная информация вместе с деревом программы сохраняется в XML-документ.

После анализа процедура сохраняется в списке обработанных процедур, для того чтобы в дальнейшем эту информацию можно было сохранить в формате XML. После этого начинается обход следующей процедуры.

7. ПРОВЕРКА КРИТЕРИЯ ПОДТВЕРЖДЕННОСТИ

Ниже будет приведено детальное описание реализации проверки критерия подтвержденности информационных потоков на основе информации, предоставляемой потоковым процессором.

При внимательном анализе формулировки критерия подтвержденности информационных потоков становится очевидным, что его суть заключается лишь в проверке того, что все аргументы любого оператора программы имеют определенное значение. Таким образом, пп. 2 и 3 критерия подтвержденности лишь определяют дополнительные варианты, когда может произойти нарушение критерия, а также определяют понятие результатов и обязательных результатов структурных операторов.

Потоковый процессор WASP предоставляет информацию об истории **def**-переменных. А именно, всегда имея **def**-переменную — аргумент оператора — можно по шкале ее статусов, упомянутой в разд 3., определить, была ли она ранее инициализирована, а также определить, является ли значение данной переменной результатом **fi**-функции или нет. Напомним, что **fi**-функция используется в потоковом процессоре для слияния значений переменных с разных ветвей исполнения программы, что в частности относится к ветвям условных операторов и операторам цикла.

Алгоритм проверки критерия подтвержденности информационных потоков сводится к следующему анализу.

При добавлении оператора в дерево программы, которое строится для проверки критерия регулярности информационных потоков, рассматривается множество **def**-переменных — аргументов данного оператора. Для каждой такой **def**-переменной проверяется ее тип. При анализе нас интересуют **def**-переменные двух типов:

a) Def-переменная простого типа — результат простого оператора;
b) Def-переменная — результат **fi**-функции. Хозяином данной **def** переменной является структура **fi**-функции, что соответствует случаю, когда переменная инициализировалась в цикле или на ветви условного оператора.

Далее, если данная **def**-переменная является переменной простого типа, то проверка шкалы статусов дает возможность определить, что переменная не была инициализирована по какому-то из вариантов исполнения программы (статус ‘**u**’ или ‘**i**’), а это является нарушением п. 1 критерия подтвержденности.

В случае, если **def**-переменная является результатом **fi**-функции, необходимо проверить наличие в шкале статусов переменной статуса ‘**p**’, что будет означать не обязательную инициализацию переменной при прохождении структурного оператора. Подобная ситуация может возникнуть, только если до структурного оператора значение переменной было не определено. Таким образом, если присутствует статус ‘**p**’, то тем самым мы сразу сталкиваемся с нарушением подпункта c) пунктов 2 и 3 критерия подтвержденности информационных потоков. При этом автоматически отсекаются проверки подпунктов a) и b).

В последнем случае остается определить тот структурный оператор, который нарушает критерий подтвержденности (оператор цикла или условный оператор), что легко определяется через структуру данных **fi**-функции.

Таким образом, можно легко определить тот оператор цикла, который не обязательно инициализирует переменную–аргумент текущего оператора. К сожалению, в случае условного оператора такая информация не доступна, и хотя она может быть обнаружена иными способами, но в силу их сложности такой анализ не производится. Также потоковый анализ предоставляет для циклов информацию о том, будет ли тело данного цикла исполняться, по крайней мере, один раз. На основе этих данных проверяется пункт 3 критерия. Для циклов while, loop анализ возможного неисполнения тела цикла не производится.

После того как нарушение критерия подтвержденности информационных потоков обнаружено, создается объект, представляющий ошибку подтвержденности соответствующего типа, и в него сохраняется вся необходимая информация. После этого ошибка сохраняется на анализируемом линейном участке программы.

Выше был подробно изложен реализованный алгоритм проверки подтвержденности информационных потоков. Как видно, проводимый анализ является контекстно-чувствительным.

8. СОХРАНЕНИЕ РЕЗУЛЬТАТОВ АНАЛИЗА В XML-ДОКУМЕНТ

После завершения работы анализатора имеется построенное дерево программы, которое содержит информацию о начальных и конечных контекстах линейных участков программы, множества аргументов и результатов для каждого из операторов. А также обнаруженные ошибки — нарушения критериев регулярности и подтвержденности информационных потоков.

Существует несколько причин, для того чтобы сохранять полученную информацию.

- Необходимость адекватной визуализации информации о найденных нарушениях критериев регулярности и подтвержденности. Оболочка XDS, в рамках которой могла бы производиться визуализация результатов, обладает очень “бедными” возможностями, которые не позволяют справиться с данной задачей.
- Возможность переиспользовать результаты анализа. В этом случае нет необходимости повторного анализа программы при условии, что ее входной текст не был модифицирован.
- Результаты анализа содержат в себе множество дополнительной информации, которая может быть использована далее.

Язык разметки документов XML [9] — это набор специальных инструкций, называемых тэгами, предназначенных для формирования в документах какой-либо структуры и определения отношений между различными элементами этой структуры. Тэги языка, или, как их иногда называют, управляющие дескрипторы, в таких документах каким-то образом кодируются, выделяются относительно основного содержимого документа и служат в качестве инструкций для программы, производящей показ содержимого документа. Контроль над правильностью использования дескрипторов осуществляется при помощи специального набора правил, называемых DTD-описаниями, которые используются при разборе документа. Для каждого класса документов определяется свой набор правил, описывающих грамматику соответствующего языка разметки.

То, что для сохранения информации был выбран формат XML, объясняется возможностью использовать следующие преимущества.

- Текстовый формат языка делает его легко понимаемым, хотя и требует большего объема дискового пространства, чем любой бинарный формат.

- Существует множество средств для работы с XML-документами. В частности синтаксические анализаторы XML-документов для различных языков программирования.
- Возможность формально описать грамматику строящихся файлов с результатами анализа.
- Как следствие предыдущих пунктов, любой разработчик может легко реализовать собственный графический интерфейс для визуализации полученных результатов.
- Возможность проверки правильности формата сохраненных данных с использованием DTD, описанной в Приложении 1, и любого инструмента проверки целостности XML-документов. Это обеспечило большое удобство при тестировании модуля сохранения аттрибутированного дерева программы в XML-документ.

Процесс сохранения информации в XML-документ реализован в двух модулях: **Linear Streams Printer**, **XML Printer**. Первый из них предназначен для обхода дерева программы, а второй для сохранения соответствующих его элементов. В целом то, что дерево программы, как и структура XML-документа, представляет собой дерево, позволило ввести естественное отображение дерева программы в XML-документ.

9. ОБЗОР РАБОТ

Точные формулировки критериев подтвержденности и регулярности были даны в статье И. В. Поттосина [1]. Существует ряд работ, которые являются её непосредственными предшественниками. В работе [10] описываются “несовершенства”, которые формально можно обнаружить в программах. Работа [4] предлагает правила, определяющие так называемую каноническую форму, автоматически исключающую ряд подобных несообразностей.

Работа [8] развивает и пополняет список излишеств, описанных в работах [4,11], и которые не должны иметь место в добротных программах. Для каждого такого излишества приводится его точное определение. Для ряда излишеств используются формализмы, предложенные в [4]. Более подробно описание критериев добротности программ было приведено в разд 2. данной работы.

Критерий регулярности информационных потоков часто может нарушаться в программах без ошибок. Тем не менее, данный критерий, цель

которого обнаруживать переплетения информационных потоков, безусловно, может служить для локализации участков входного текста программы, которые потенциально могут содержать ошибки вычислений.

Критерий подтвержденности информационных потоков позволяет выявить трудно отлавливаемые ошибки использования неинициализированных переменных. Что касается аналогов, то в настоящий момент использование неинициализированных переменных проверяется многими компиляторами. Например, JBuilder 6.0 с jdk1.3 предоставляет подобную диагностику, хотя и не столь полную, как та, что может быть получена построенным анализатором. В частности, не предоставляется информация о том, какой оператор производит необязательную инициализацию переменных, использующихся в дальнейшем в качестве аргументов. Другим аналогом является анализатор WASP семантических ошибок периода исполнения, несмотря на то, что он нацелен на поиск несколько иных ошибок, например, таких как выход за границы массива, хотя общей является проверка на использование не инициализированных переменных. Следует отметить, что построенный анализатор выявляет не просто использования не инициализированных переменных, но обнаруживает отсутствие инициализации переменных на всех путях исполнения программы (ветви условных операторов, тела циклов).

Существует множество программ анализаторов, таких как Extended Static Checking for Java и DevPartner Studio, которые предоставляют различную информацию на основе статического анализа входного текста программы. В основном результатами анализа таких систем являются предупреждения о возможных ошибках, таких как отсутствие инициализации используемых переменных или недостижимость определенных ветвей исполнения программы и т.д. Часто это коммерческие продукты, которые распространяются в рамках больших систем разработки программного обеспечения, и таким образом, могут быть доступны только профессиональным программистам. Существующие же не коммерческие продукты либо вовсе не обладают графическими интерфейсами, либо предоставляют минимальные возможности, такие как вывод списка предупреждений с подсветкой соответствующих строк входного текста анализируемой программы.

Созданная в рамках данной работы графическая оболочка является прототипом. По своей функциональности она может сравниваться с графическими оболочками таких систем, как JBuilder(Borland), NetBeans (Sun) или Visual Studio (Microsoft). Критериями данного сравнения служили наличие дерева программы, предоставляющего возможности навигации, возмож-

ность одновременно иметь несколько открытых файлов входного текста проекта, а также степень соответствия функциональности, реализованной в системе и её изначального предназначения. Следует отметить, что разработанная оболочка имеет, безусловно, более мощную поддержку визуализации и локализации недобротностей в программе, чем рассмотренные системы.

10. ЗАКЛЮЧЕНИЕ

Данная работа описывает реализацию программного процессора, обеспечивающего анализ добротности информационных потоков в программах на языках Модула-2/Оберон-2 на основе данных контекстно-чувствительного потокового анализа программ с аппроксимацией обязательных информационных связей.

В результате работы проведена реализация критериев добротности информационных потоков на основе данных потокового анализа, разработан визуализатор, который обладает функциональной полнотой в смысле предоставления разнообразного набора функциональных возможностей. Также разработан механизм передачи информации от анализатора добротности информационных потоков к графическому интерфейсу пользователя при помощи сохранения данных в формате XML. Формат данных XML-документа задается построенной DTD-грамматикой.

Одним из направлений развития данной работы является адаптация критериев добротности информационных потоков для объектно-ориентированных языков программирования.

В частности, планируется реализовать проверку данных критериев для языка программирования C# фирмы Microsoft. Предполагается, что реализация проверки данных критериев будет включена в создаваемую систему верификации программ на языке C#, при этом потребуется проверять критерии на основе иных данных о программе, которые могут быть доступны в рамках системы верификации, что в свою очередь представляется отдельной и интересной задачей.

СПИСОК ЛИТЕРАТУРЫ

1. **Поттосин И.В.** Добротность программ и информационных потоков // Открытые системы. — 1998. — N 6. — С. 41–45.

2. **Куксенко С.В., Шелехов В.И.** Статический анализатор семантических ошибок периода исполнения // Программирование. — 1998. — № 6. — С. 23–43.
3. **С Черноножкин.** Меры сложности программ // Системная информатика. — 1996. — Вып. 5. — С.188–227.
4. **Pan S., Dromey R.G.** Beyond Structured Programming // Proc. of the 18th Intern. Conf. on Software engineering (ICSE-18). — Berlin, 1996. — P. 268–278.
5. **Cousot P., Cousot R.** Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoint // Proc. of the 18th ACM Symposium on Principles of Programming Languages. — ACM Press, 1977. — P. 55–56.
6. **Шелехов В. И.** Структура программы в языково-ориентированном потоковом анализе // Программирование. — 1996. — № 3. — С. 47–59.
7. **Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadec F.K.** Efficiently computing static single assignment from and the control dependence graph // ACM Trans. of Programing Languages and Systems. — Vol. 13, Issue 4. — 1991. — P. 451–490
8. **Поттосин И. В.** «Хорошая программа»: попытка точного определения понятия // Программирование. — 1997. — № 2. — С. 3–17.
9. **Extensible** Markup Language (XML) 1.0 & Data Type Definition. — available at `www.w3.org`
10. **Halstead M.** Elements of Software Science. — N.Y.: Elsevier, 1977. (Русск. перевод: Холстед М.Х. Начало науки о программах. — М.: Финансы и Статистика, 1981).
11. **Kasjanov V.N., Pottosin I.V.** Application of optimization techniques to correctness problems // Construction Quality Software. — North-Holland, 1978. — P. 237–248.

1. DTD-ГРАММАТИКА ДЛЯ XML-ДОКУМЕНТА

```
<?XML encoding="UTF-8"?>
<!ELEMENT position ANY>
<!ATTLIST position file_name CDATA  #REQUIRED line CDATA #RE-
        QUIRED inline_pos CDATA #REQUIRED>
<!ELEMENT error_position (position)>
<!ATTLIST error_position type (error|operator) #REQUIRED>
<!ELEMENT substitutor ANY>
<!ATTLIST substitutor name CDATA #REQUIRED status CDATA #RE-
        QUIRED
defnumber CDATA #REQUIRED id CDATA #REQUIRED>
<!ELEMENT variable (substitutor*)>
<!ATTLIST variable name CDATA #REQUIRED status CDATA #RE-
        QUIRED defnumber CDATA #REQUIRED
id CDATA #REQUIRED>
<!ELEMENT arguments (variable*)>
<!ELEMENT results (variable*)>
<!ELEMENT context (variable*)>
<!ATTLIST context type (begin_context|end_context) #REQUIRED>
<!ELEMENT proc_call ANY>
<!ATTLIST proc_call status CDATA #REQUIRED id CDATA #REQUIRED
        number CDATA #REQUIRED>
<!ELEMENT operator (posi-
        tion,proc_call*,operator*,proc_call*,arguments,results,branch*)>
<!ATTLIST operator number CDATA #REQUIRED type (sim-
        ple_operator|if|case|for|cycle|call) #REQUIRED status CDATA
        #IMPLIED once (true|false) #IMPLIED>
<!ELEMENT stream ANY>
<!ATTLIST stream content CDATA #REQUIRED type (informa-
        tion_stream_realization | independent_information_stream | inter-
        section_r_i_stream | information_stream_s_n | informa-
```

tion_stream_s_m | trun-
cated_realization_n_to_m|set_of_error_vertexes) *#REQUIRED*>
<*!ELEMENT* **regular_error (stream*)**>
<*!ATTLIST* **regular_error message** *CDATA #REQUIRED* **type (A|B|U)** *#RE-QUIRED*>
<*!ELEMENT* **confirmation_error (error_position*)**>
<*!ATTLIST* **confirmation_error message** *CDATA #REQUIRED* **type (A|B|C|D)** *#REQUIRED* **name** *CDATA #REQUIRED*
status *CDATA #REQUIRED* **defnumber** *CDATA #REQUIRED* **id** *CDATA #REQUIRED*>
<*!ELEMENT* **branch (con-**
text*,confirmation_error*,regular_error*,operator*)>
<*!ELEMENT* **procedure (position,branch)**>
<*!ATTLIST* **procedure name** *CDATA #REQUIRED* **catalogid** *CDATA #RE-QUIRED* **position** *CDATA #IMPLIED*>
<*!ELEMENT* **structure (procedure*)**>

2. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС

Рис. 1. Графический интерфейс

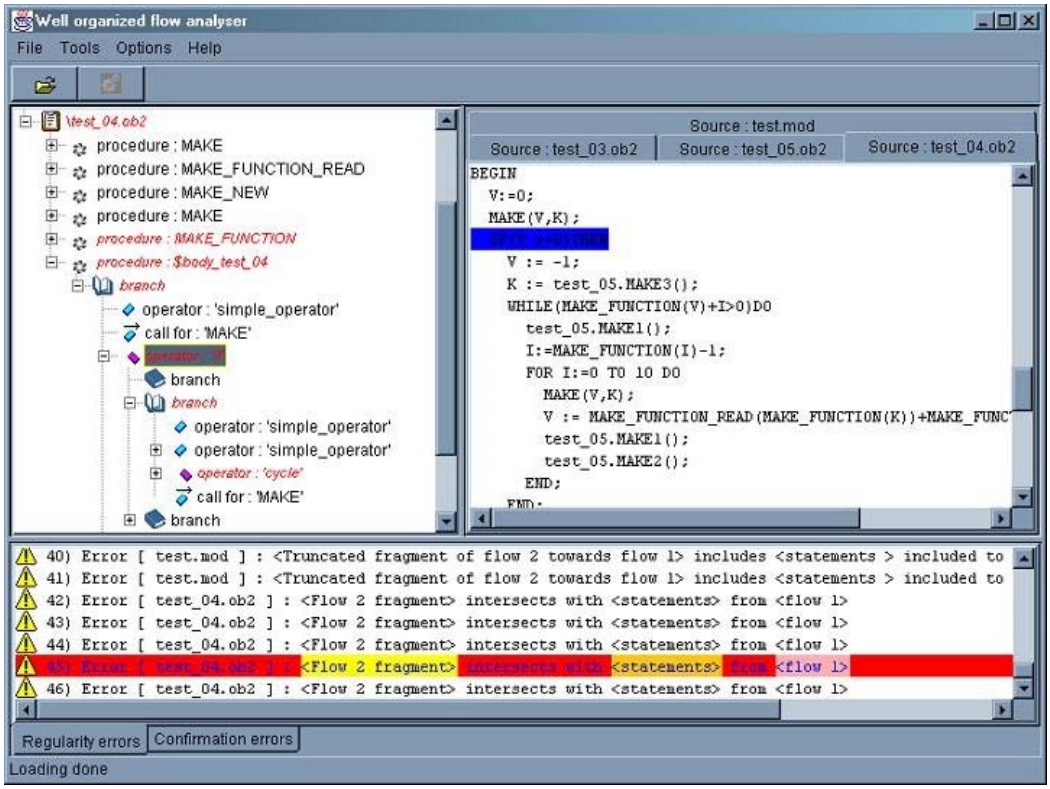


Рис. 2. Навигация по дереву программы

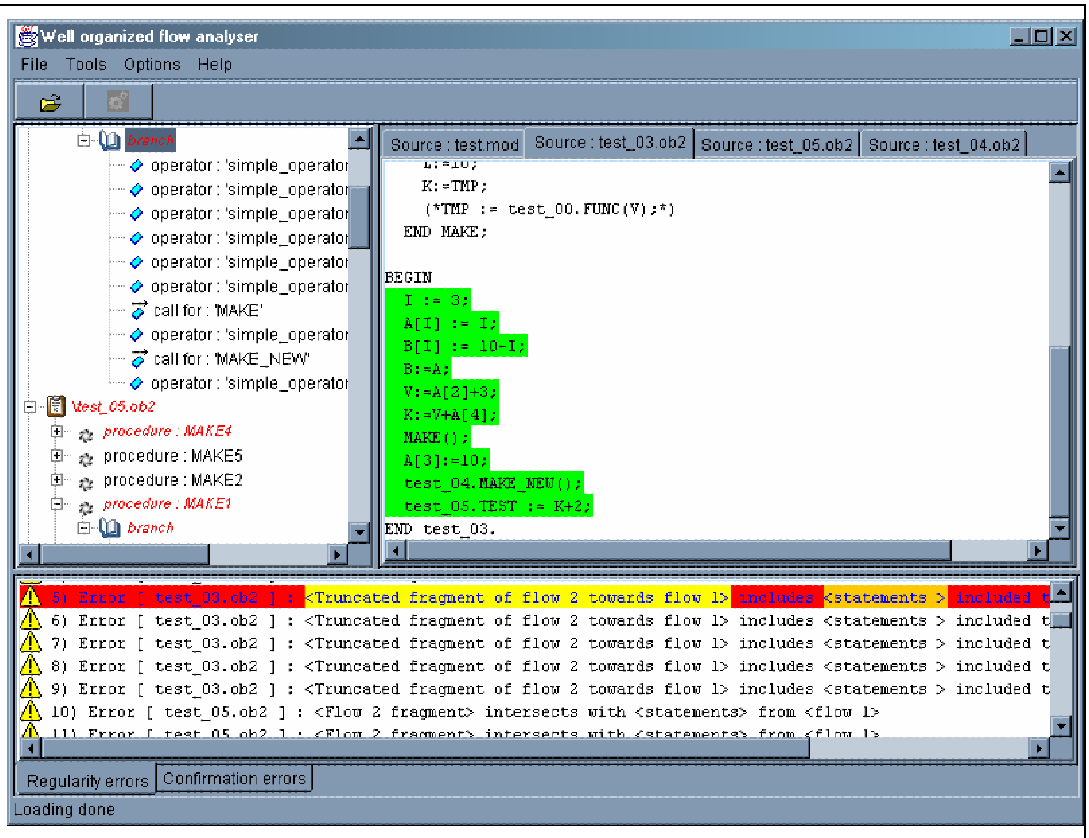


Рис. 3. Визуализация ошибок Регулярности информационных потоков (1)

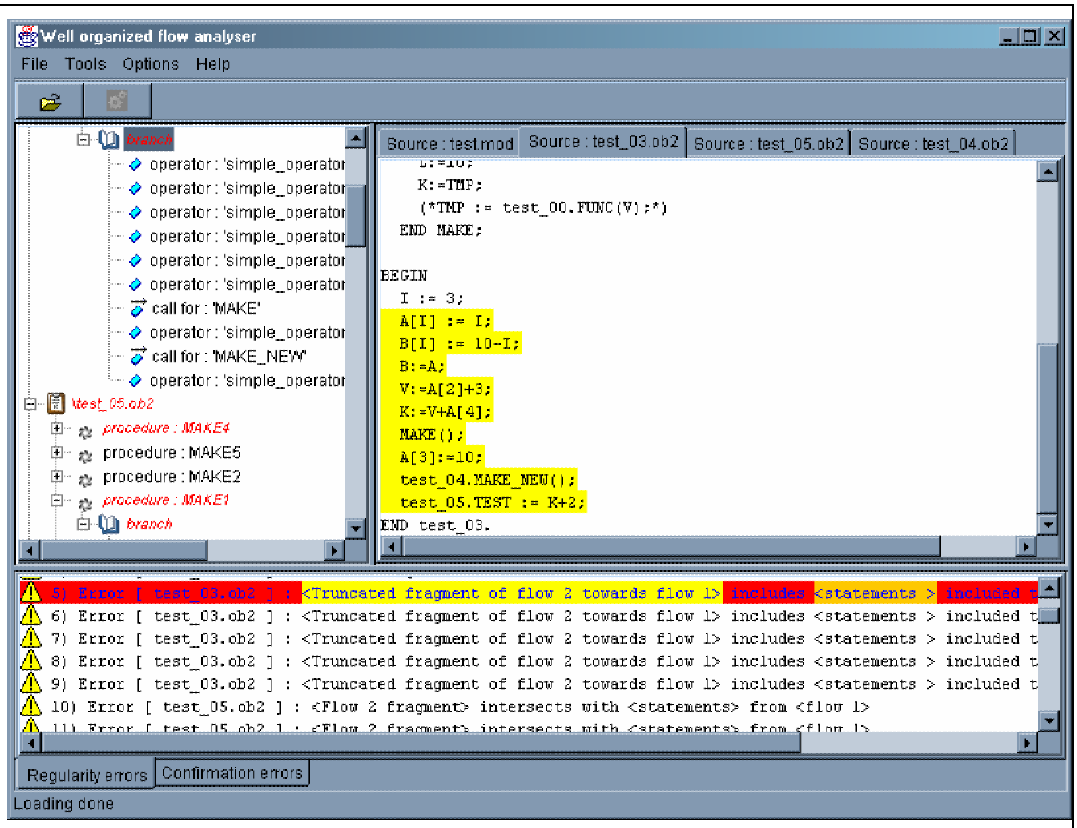


Рис. 4. Визуализация ошибок Регулярности информационных потоков (2)

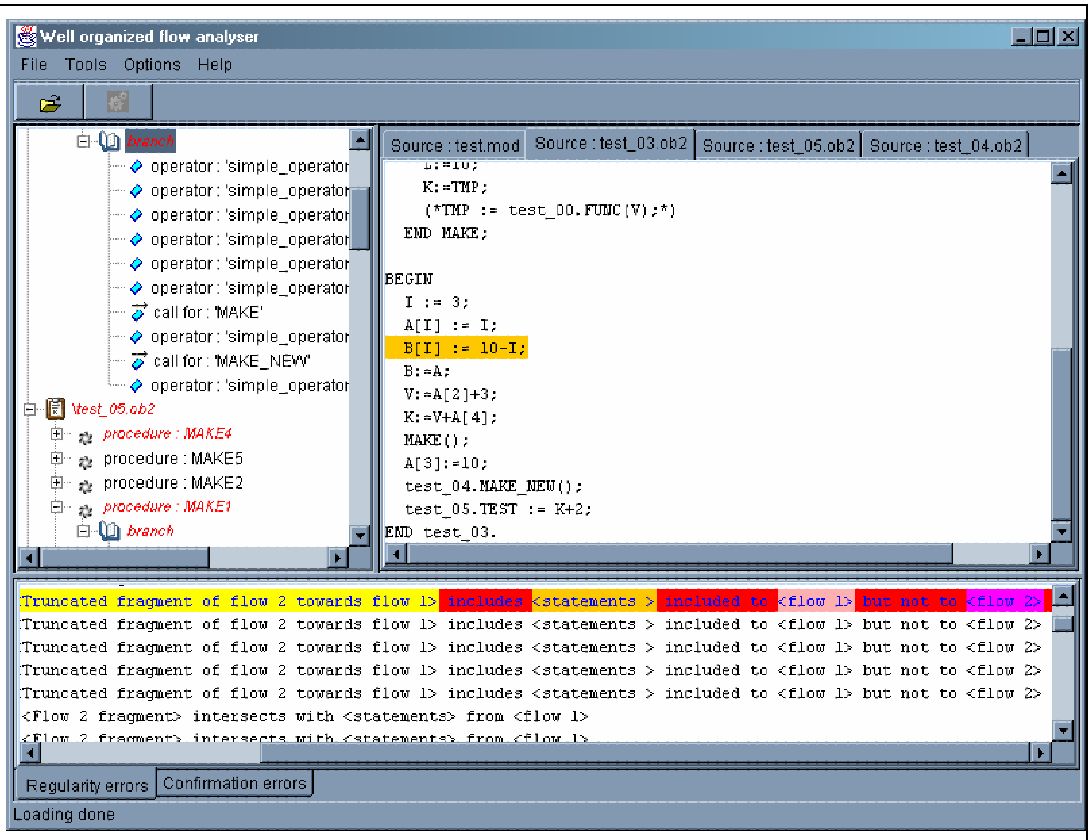


Рис. 5. Визуализация ошибок Регулярности информационных потоков (3)

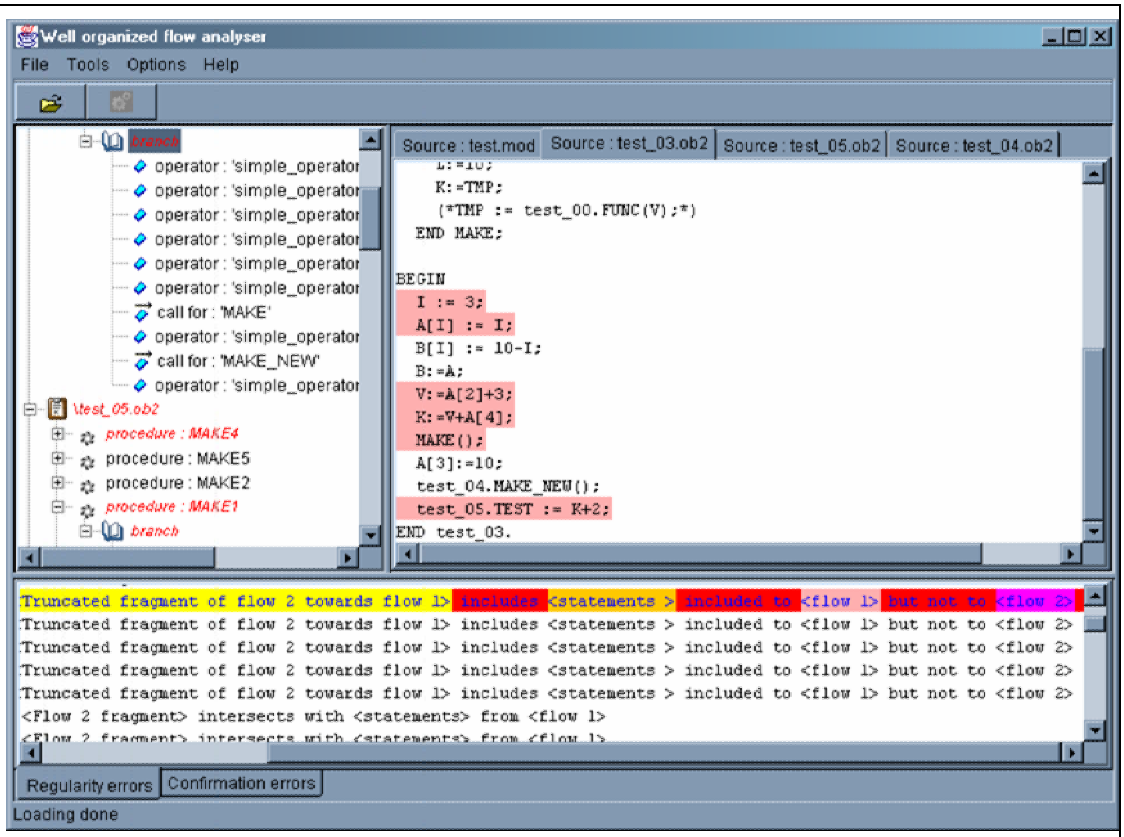


Рис. 6. Визуализация ошибок Регулярности информационных потоков (4)

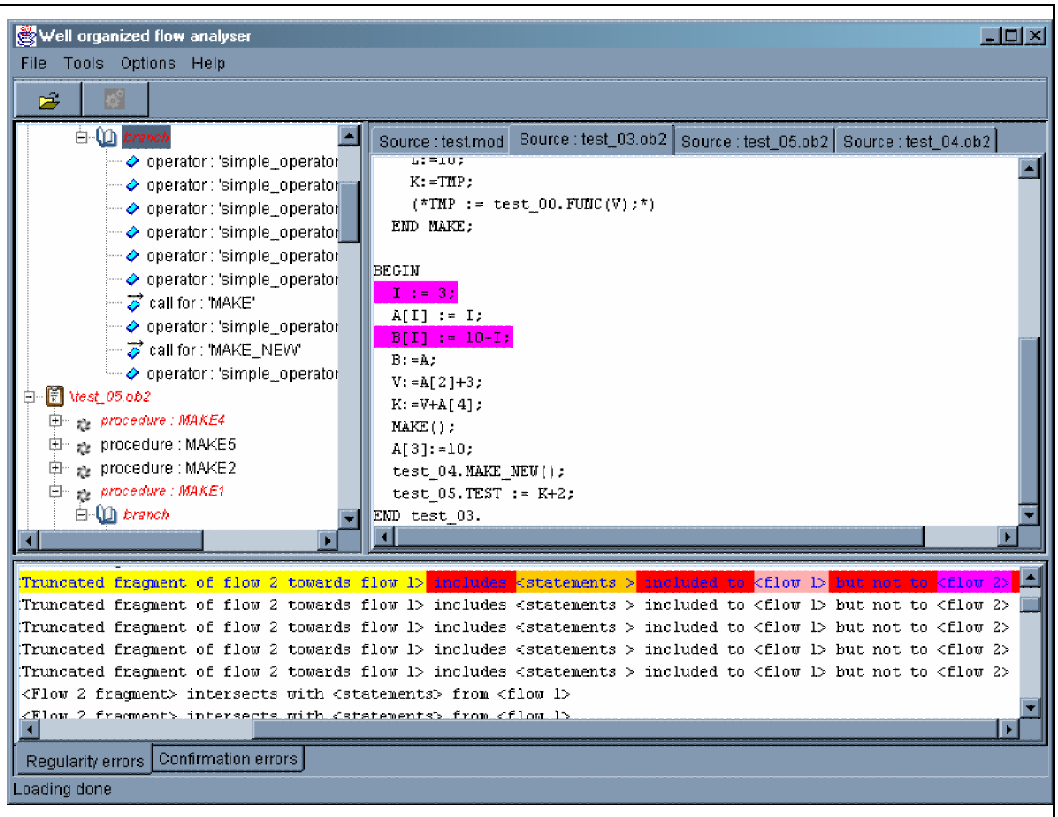


Рис. 7. Визуализация ошибок Регулярности информационных потоков (5)

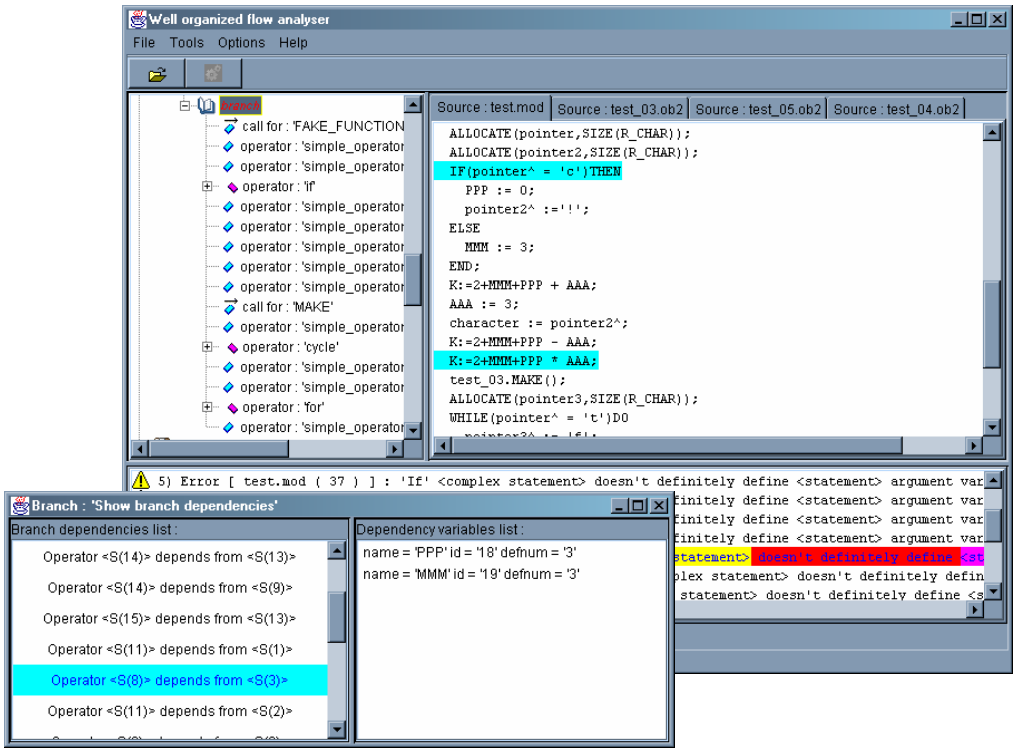


Рис. 8. Визуализация дополнительной информации (Зависимости)

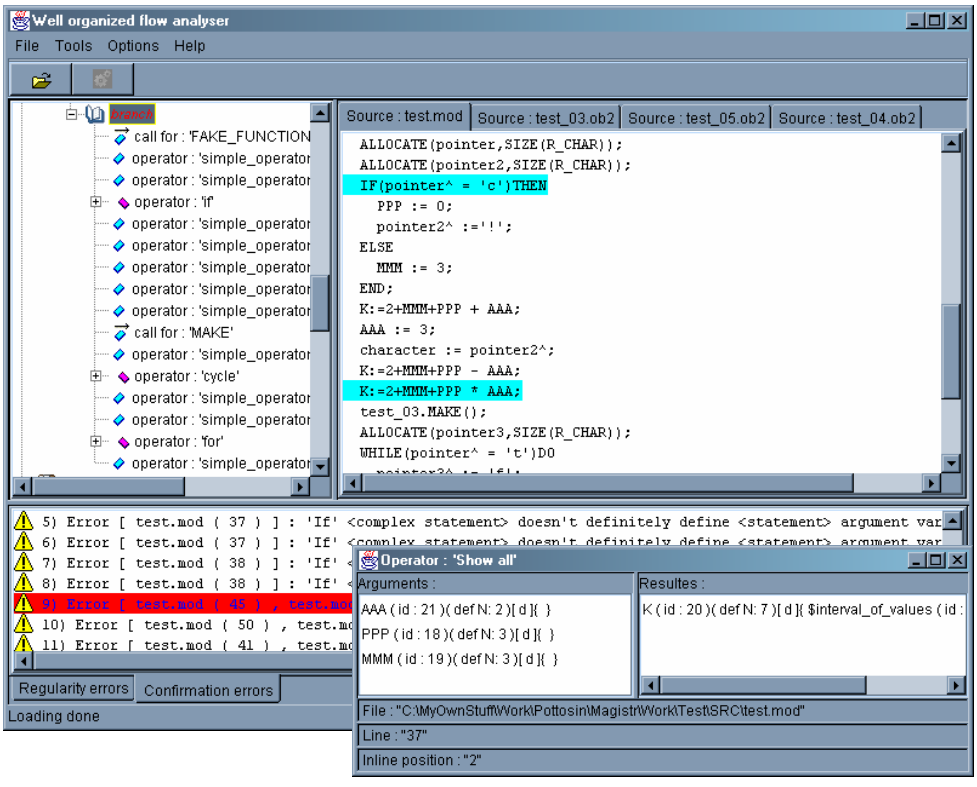


Рис. 9. Визуализация дополнительной информации (Контексты)

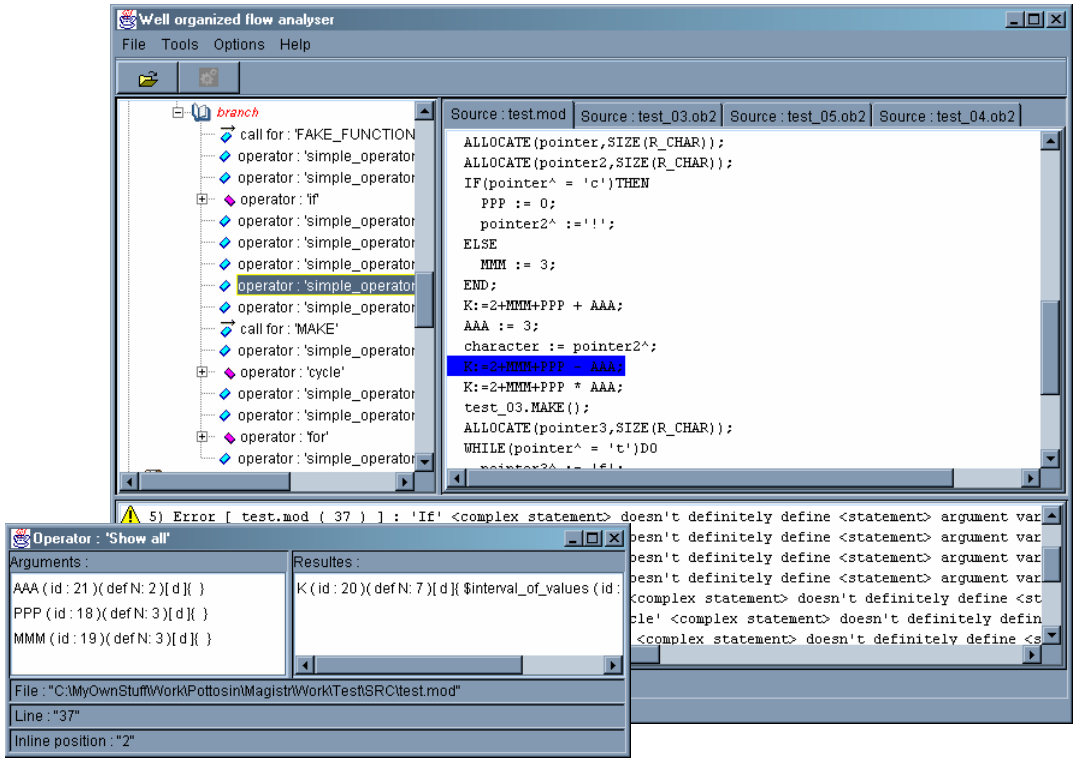


Рис. 10. Визуализация дополнительной информации (Атрибуты оператора)

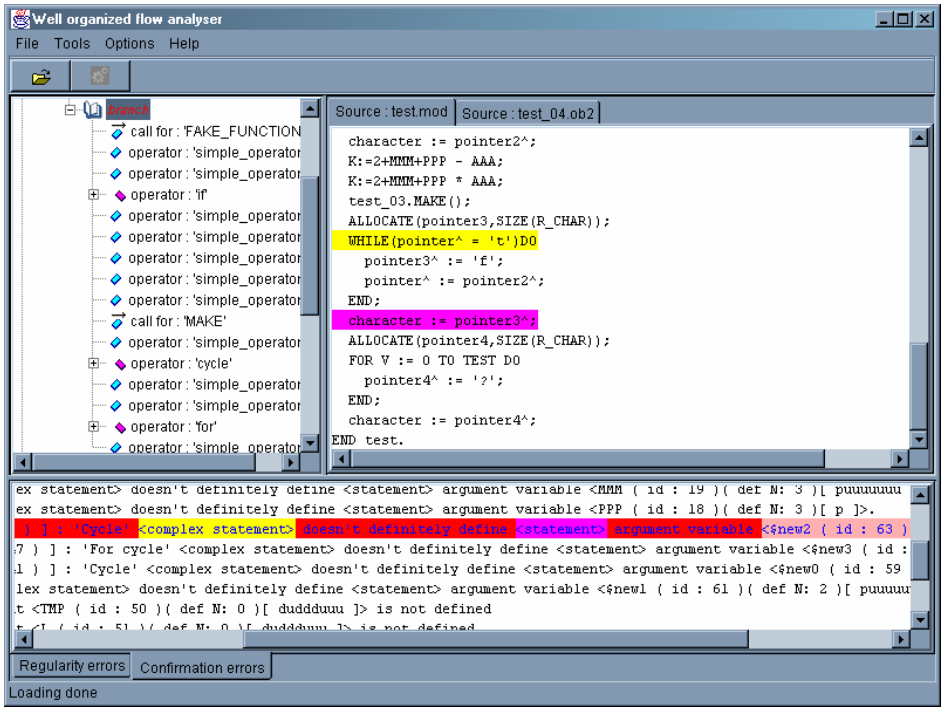


Рис. 11. Визуализация ошибок Подтвержденности информационных потоков

И. С. Запреев

АНАЛИЗ ДОБРОТНОСТИ ИНФОРМАЦИОННЫХ ПОТОКОВ В ПРОГРАММАХ НА ЯЗЫКАХ МОДУЛА-2/ОБЕРОН-2

Препринт
111

Рукопись поступила в редакцию 19.12.03
Редактор З. В. Скок
Подписано в печать 5.03.04
Формат бумаги 60 × 84 1/16 Объем 2.7 уч.-изд.л., 3.0 п.л.
Тираж 50 экз.

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383-2) 34-22-02