Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Сибирский федеральный университет»

На правах рукописи

Удалова Юлия Васильевна

ОТЛАДКА И ВЕРИФИКАЦИЯ ФУНКЦИОНАЛЬНО-ПОТОКОВЫХ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

05.13.11 – «Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей»

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата технических наук

Научный руководитель
Легалов А.И.

Красноярск – 2014

ОГЛАВЛЕНИЕ

Введение

Разработка параллельных программ является нетривиальным процессом независимо от используемой парадигмы программирования. Наряду с необходимостью анализа корректности логики выполнения программы, в большинстве систем параллельного программирования также приходится решать задачу распределения ресурсов, используемых для выполнения процессов. Ориентация на инструментальные средства, обеспечивающие динамическое распределение ресурсов, все равно оставляет актуальным анализ корректности создаваемой программы, для проведения которого необходимо использование средств отладки и верификации.

Отладка программ в настоящее время хорошо проработана для последовательного программирования. Существует множество отладчиков, поддерживающих разнообразные приемы, например, пошаговый анализ кода с отображением состояния переменных, пропуск уже отлаженных фрагментов программ, отладку только требуемых функций или процедур, удобный просмотр кода, проверку логических условий пользователя над переменными программы.

Отладка параллельных программ сопровождается дополнительными трудностями, поскольку в ходе нее необходимо учитывать наличие нескольких одновременно исполняемых потоков команд и специфику их взаимодействия, для чего необходимо значительно расширить инструментарий среды отладки. В частности, требуется другой пользовательский интерфейс, предоставляющий удобную навигацию по коду программы, необходимы дополнительные функции, позволяющие анализировать конфликтные и тупиковые ситуации. В настоящее время достигнуты определенные успехи в разработке соответствующих средств [2, 24, 41, 44, 53, 54, 95, 96, 98, 102, 103, 105] для систем, использующих потоки или процессы, расширяющие технику последовательного выполнения программ.

Верификация [27, 57, 58, 59, 80, 113] позволяет установить корректность программы [3, 45, 75, 112] на более формальном уровне, чем отладка. В процессе верификации подтверждаются или опровергаются различные свойства,

определяющие корректность написанного кода, относительно заданной разработчиком спецификации [8, 13, 42, 79] или общие для класса рассматриваемых программ [81-83], например, такие как выход за границы типов или массивов. Методы верификации делятся на методы доказательства теорем [28, 35, 76, 88, 89, 116, 129] и методы проверки моделей [46, 47, 132].

Направление верификации программ интенсивно развивается. Имеются работы по верифицирующим компиляторам [50, 104]. Реализованы системы автоматического доказательства теорем, такие как PVS [20] и Isabelle [19, 131]. Создано значительное число верификаторов, применяющих методы проверки моделей, например, верификаторы многопоточных параллельных моделей Spin [22, 130] и Bogor [18]. Развивается направление верификации автоматных программ с использованием методов проверки моделей [15, 16, 25, 36, 56].

Сложность верификации параллельных программ заключается в огромном количестве вариантов взаимодействия процессов в реальном времени, при котором наряду с логическими ошибками появляются и ошибки, связанные с последовательностью исполнения параллельных фрагментов. Ошибки межпроцессного взаимодействия, ошибки синхронизации параллельных процессов, конфликты, возникающие при одновременном обращении нескольких процессов к одним и тем же общим ресурсам, составляют дополнительные препятствия, как для достижения корректности программы, так и для ее завершения.

Стремление к эффективному использованию архитектур современных параллельных вычислительных систем (ПВС) ведет к тому, что в создаваемых программах производятся не только вычисления, определяемые решаемой задачей, но и осуществляется планирование вычислительных ресурсов, направленное на ускорение и балансировку вычислений [60, 61], а также рациональное использование памяти и коммуникационных каналов. При обмене данными между одновременно выполняющимися процессами применяются плохо контролируемые асинхронные методы, что легко может привести к конфликтным и тупиковым ситуациям. Подобные проблемы возникают при использовании

практически всех современных систем параллельного программирования [6, 7, 9, 14, 26, 30, 48, 52, 55, 77, 78, 94, 111, 114, 115, 125], независимо от того, используют ли они механизмы передачи сообщений или взаимодействие через общую память.

Положение усугубляется тем, что наличие разнообразных архитектур параллельных вычислительных систем (ПВС) ведет как к разнообразию методов написания параллельных программ, так и разным способам отладки и верификации. Это требует изучения новых инструментальных средств при переходе с одной ПВС на другую. Широкое распространение гетерогенных архитектур еще больше запутывает ситуацию с разработкой, отладкой и верификацией параллельных программ.

Попытки избавиться от ресурсной зависимости при разработке прикладных параллельных программ привели к созданию архитектурно-независимых языков параллельного программирования, которые вместо императивного стиля обычно базируются на функциональной или функционально-потоковой [10, 33, 63, 68-74, 108, 109, 122, 123] парадигме программирования и управлении по готовности данных. Применение подобных языков позволяет создавать программу без явного указания управляющих воздействий между выполняемыми действиями, так как последние могут автоматически выстраиваться по сформированным в ходе написания кода информационным зависимостям [106, 107], наличие которых является необходимым при любом стиле программирования. Помимо этого, применение функционального параллельного программирования позволяет абстрагироваться от архитектурных и ресурсных особенностей вычислительных систем и сосредоточиться при написании кода только на особенностях предметной области решаемой задачи [67]. Написанная таким образом программа требует верификации и отладки только информационных зависимостей. Предполагается, что последующая привязка к реальным архитектурам будет осуществляться на более поздних этапах с использованием дополнительных инструментальных средств (трансляторов, загрузчиков и др.). Допустимо также ручное преобразование таких программ в архитектурно зависимый код. Это

позволяет на первом этапе сконцентрироваться на логике приложения и написании программы с максимальным параллелизмом. Разбиение процесса разработки на два этапа облегчает создание функционала, отвечающего за предметную область, и позволяет на втором этапе выстраивать только архитектурную привязку, при которой уже разработанный функционал является логически корректным.

Применение функционально-потокового подхода повышает эффективность процесса разработки программного обеспечения, но, вместе с тем, предъявляет к нему на каждом из рассмотренных этапов свои специфические требования. С одной стороны, создание архитектурно-независимых параллельных программ позволяет формировать библиотеки и пакеты, которые в дальнейшем, с использованием средств архитектурной привязки, могут переноситься на различные вычислительные системы. С другой стороны, к методам верификации и отладки архитектурно-независимых параллельных программ предъявляются требования, не совпадающие с теми, что используются в существующих инструментальных средствах.

Функционально-потоковая модель вычислений определяет программу как ациклический граф [106, 107] с вершинами-операторами и информационными связями между ними. Это позволяет использовать граф программы для выделения анализа путей передачи данных и визуализации процессов отладки и верификации [119-121]. Отсутствие механизмов передачи сообщений и разделяемой памяти позволяет отбросить анализ соответствующих конфликтов. Выполнение программы на «неограниченных ресурсах» [65, 66] позволяет не рассматривать ресурсные конфликты. В результате отладка и верификация концентрируются на логике выполнения программы, при этом наличие зависимостей только по данным дает возможность анализировать логику программы одновременно для нескольких информационно несвязанных ее операторов.

Существующие инструментальные средства [31, 32, 97, 99, 100, 133] обеспечивают выполнение и отладку функциональных и функционально-

потоковых параллельных программ, предоставляя различные возможности проверки:

- автоматический или управляемый пользователем обход программы при отладке;
- расстановку контрольных точек, выделение ошибочных вычислений, визуализацию информационного графа программы;
- возможность отладки частично не законченных программ;
- прямой и обратный ход отладки.

Но отладка функционально-потоковых параллельных программ может быть дополнена, например, такими возможностями, как введение различных шагов отладки, связанных с порядком обхода графа программы, использование отладочных переменных и пользовательских условий.

Инструментальные средства, обеспечивающие верификацию программ, представлены преимущественно двумя типами верификаторов: основанные на методах проверки моделей, обрабатывающие заданный набор параллельных или асинхронных команд [18, 22, 130] и применяющие методы доказательства теорем [19, 20, 131], анализирующие преимущественно последовательные программы.

Существующие автоматические верификаторы ориентированы в основном на последовательные, асинхронные, многопроцессные или многопоточные программы. Верификация функциональных программ возможна в нескольких существующих инструментальных средах [19]. Возможности верификации функционально-потоковых параллельных программ не представлены.

Таким образом, возможность расширения существующего инструментария для отладки и отсутствие инструментальных средств верификации определяют актуальность исследования и разработки методов верификации и отладки архитектурно-независимых функционально-потоковых параллельных программ.

Целью работы является разработка методов и инструментальных средств, обеспечивающих отладку и верификацию функционально-потоковых параллельных программ.

Для достижения указанной цели в работе решаются следующие задачи:

– анализ существующих методов отладки и верификации параллельных программ, выбор методов эффективных для функционально-потоковой парадигмы программирования;

– разработка методов, обеспечивающих отладку и верификацию функционально-потоковых параллельных (ФПП) программ;

– создание инструментальных средств, обеспечивающих поддержку предложенных методов отладки и верификации.

Предмет и объект исследования. Объектом исследования является функционально-потоковая модель параллельных вычислений. Предмет исследования – методы отладки и верификации ФПП программ.

Методы исследования. Поставленные задачи решались посредством методов доказательства теорем, формального доказательства правильности программ, методов проверки моделей, элементов теории графов, теории языков программирования, математической логики и интервального анализа.

При создании программных инструментов для отладки и верификации использовались структурное и объектно-ориентированное программирование.

Положения, выносимые на защиту.

1. Методы и алгоритмы обхода траекторий отладки функционально-потоковых параллельных программ.

2. Метод верификации функционально-потоковых параллельных программ, использующий информационные графы программы, спецификацию пользователя и интервальные константы.

3. Метод верификации функционально-потоковых параллельных программ с асинхронными списками.

4. Архитектура программной среды, обеспечивающей поддержку разработанных методов отладки и верификации функционально-потоковых параллельных программ.

Научная новизна.

1. Предложен метод отладки функционально потоковых параллельных программ, базирующийся на различных принципах обхода траекторий отладки,

позволяющий в сочетании с методами визуализации отладочной информации повысить эффективность процесса поиска и анализа ошибок, как в текстовом, так и графическом режимах представления функционально-потоковых параллельных программ. Некоторые принципы обхода траекторий отладки и элементы применения графов встречаются в отладчиках ФПП программ, но представленный комплекс режимов отладки уникален.

2. Разработан метод верификации функционально-потоковых параллельных программ, основанный на принципе обхода информационного графа с использованием интервальных констант, позволяющий осуществлять автоматизированную проверку спецификации пользователя и получить интервальные оценки результатов работы операторов. Метод опирается на известные методы доказательства теорем, изначально сформулированные для императивных последовательных программ. Применение методов доказательства теорем к ФПП программе в совокупности со спецификацией данных ФПП программы на графе с помощью интервальных констант оригинальны.

3. Разработан метод верификации функционально-потоковых параллельных программ с асинхронными списками, базирующийся на принципе комбинаторного анализа результатов выполнения программных операторов, позволяющий формировать оценочный пакет результатов выполнения программ с асинхронными списками. Базой метода являются общие принципы известного метода проверки модели, применение его к верификации ФПП программ с асинхронными списками оригинально.

Методы отладки и верификации разработаны с учетом применения к функционально-потоковому параллельному языку Пифагор, для которого до получения результатов диссертационной работы был реализован только простейший отладчик, а способов автоматической верификации не существовало.

Практическая значимость.

1. На основе предложенных методов отладки и верификации разработаны инструментальные средства, интегрированные в существующую систему функционально-потокового параллельного программирования Пифагор.

2. Результаты работы использовались при выполнении:

– научно-исследовательских работ в рамках федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» по теме «Инструментальная поддержка архитектурно-независимой разработки параллельных программ на основе функционально-потоковой парадигмы параллельного программирования», выполняемой в 2012-2013 гг.;

– научно-методического проекта «Среда разработки для языка параллельного программирования Пифагор» в рамках «Программы развития СФУ на 2007–2010 годы».

3. Полученные научные результаты использованы в учебном процессе по дисциплине «Технология программирования» в виде лекций по этапам разработки программного обеспечения для подготовки специалистов по специальности 230101.65 «Вычислительные машины, комплексы, системы и сети» в ФГАОУ ВПО «Сибирский федеральный университет».

Все результаты практического применения диссертационного исследования подтверждены соответствующими актами и свидетельствами о регистрации программного обеспечения.

Достоверность полученных результатов подтверждается корректным использованием математического и программного аппарата, работоспособностью разработанного программного обеспечения (ПО), применением разработанного ПО в научной деятельности по развитию представленных методов отладки и верификации и в учебном процессе.

Апробация работы. Основные положения диссертации докладывались и обсуждались на следующих конференциях и семинарах: четвертой Российско-германской школе по параллельным вычислениям, г. Новосибирск, 2007г.; Всероссийских научно-технических конференциях «Молодежь и наука –XXI век», г. Красноярск, 2007-2009гг.; четвертой Сибирской школе-семинаре по параллельным вычислениям, г. Томск, 2007г.; Международной научной конференции «Параллельные вычислительные технологии (ПаВТ'2009)», г. Красноярск, 2009г.; XI Всероссийской научно-практической конференции

«Проблемы информатизации региона (ПИР-2009)», г. Красноярск, 2009г.; Международной Ершовской конференции по Информатике, рабочий семинар «Наукоемкое программное обеспечение», г. Новосибирск, 2011г.; Международной суперкомпьютерной конференции «Научный сервис в сети Интернет: все грани параллелизма», Новороссийск, 2013г.

Личный вклад автора в научные работы, опубликованные в соавторстве с научным руководителем, заключается в следующем: анализе существующих методов отладки и верификации на предмет применения к ФПП языку; разработке методов и алгоритмов отладки и верификации ФПП программ; представлении архитектуры инструментальной среды для разработки ФПП программ. Совместно с научным руководителем автор осуществлял постановку целей и задач, выбор методов для реализации, анализ полученных результатов. В совместных публикациях автора с Сиротининой Н.Ю. и Кропачевой М.С. включены разработанные ими теоретические идеи о верификации ФПП программ на языке Пифагор, являющиеся параллельной научной разработкой и не задействованные в диссертационной работе автора. В совместных публикациях автора с Матковским И.В. и Васильевым В.М., научные результаты автора являются прикладным дополнением к концепции разрабатываемого ими транслятора ФПП программ.

Публикации. По теме диссертации опубликовано двадцать научных работ, из которых три статьи в изданиях, рекомендуемых ВАК и два свидетельства о государственной регистрации программ для ЭВМ.

Структура и объем работы. Диссертация состоит из введения, пяти глав, заключения и трех приложений. Работа содержит сто сорок страниц основного текста, сорок девять рисунков и две таблицы. Список литературы содержит сто тридцать четыре наименования.

Во введении приводится общая характеристика работы и дается краткий обзор содержания диссертации.

В первом разделе проведен анализ применяемых методов отладки для различных параллельных вычислительных систем, на основе которого составлена классификация существующих подходов, включающая такие критерии как:

ориентация на параллельные примитивы, прохождение параллельной программы, выбор шага отладки, виды контрольных точек, использование отладочных значений, направление хода отладки, графическое представление процесса отладки.

Критерии представленной классификации рассмотрены с учетом их возможного применения к функционально-потоковой параллельной модели вычислений. Предложены режимы выбора шага отладки, основанные на различных способах обхода информационного графа ФПП программы. Описаны особенности применения точек останова и наблюдения к ФПП программам. Для визуализации отладки ФПП программ предлагается использовать ее информационные графы, позволяющие выделять как уже отработанные, так и неправильно вычисленные вершины-операторы, а также добавлять отладочные выражения (пользовательские условия).

Во втором разделе проведен анализ методов верификации программ, рассмотрены возможности существующих инструментальных средств верификации как последовательных, так и параллельных программ. Проводится исследование того, каким образом существующие методы верификации могут быть использованы совместно с функционально потоковой парадигмой параллельного программирования. Выделяются методы, целесообразные для верификации ФПП программ. Выбирается форма спецификации ФПП программ. Задача верификации функционально-потоковой параллельной программы формулируется как верификация на информационном графе программы, что позволяет сужать множество рассматриваемых утверждений.

В третьем разделе описаны разработанные методы отладки функционально-потоковых параллельных программ и связанный с ними процесс визуализации результатов отладки. На основе проведенного в первом разделе анализа выделяются четыре режима отладки: пошаговый режим, режим отладки слоев, режим отладки ветвей, режим проверки формул. Представлены различные способы выполнения и визуализации процесса отладки в различных режимах визуализации: режим операторов, список функций, свертка функций,

отображение на графе программы. Предложены элементы архитектуры среды, обеспечивающей отладку функционально-потоковых параллельных программ.

В четвертом разделе описаны разработанные методы верификации функционально-потоковых параллельных программ. Рассмотрены возможности проверки программы без ее выполнения и при отсутствии спецификации пользователя. Предложены:

- метод верификации функционально-потоковых параллельных программ с асинхронными списками, основанный на переборе возможных путей выполнения программы;
- метод формальной верификации, основанный на спецификации входных данных программы с помощью конкретных значений и дополнительных предложенных формул и добавлении условий к вычислениям программы на ее информационном графе.

Описаны формы представления процесса верификации пользователю, среди которых присутствует верификация на графе программы, возможная благодаря модели функционально-потоковых параллельных вычислений. Отображена часть особенностей реализации предложенных методов верификации в инструментальной среде.

В пятом разделе представлена архитектура программных модулей среды для разработки, отладки и верификации функционально-потоковых параллельных программ. Описаны принципы работы транслятора, отладчика и верификатора, дано словесное описание их базовых алгоритмов, представлен интерфейс разработанной среды.

В заключении формулируются основные результаты работы.

В приложении А представлены примеры отладки ФПП программ в разработанной инструментальной среде.

В приложении Б приведены правила и примеры обработки формул спецификации, используемых для верификации ФПП программ.

В приложении В представлены копии свидетельств и справок о внедрении результатов диссертационной работы.

1 Инструментальные средства для отладки параллельных программ

1.1 Ошибки в параллельных программах

В параллельной программе наряду с синтаксическими и логическими ошибками, характерными для последовательных программ, возникают ошибки, связанные с особенностями организации параллельных вычислений. Параллельная программа может быть реализована в рамках одного или нескольких параллельных примитивов в зависимости от способа ее описания:

– в процессе создания программы с явным параллелизмом разработчик оперирует процессами или потоками (нитями), при этом распределение ресурсов осуществляется явно, как правило, на этапе написания программы;

– программа с неявным параллелизмом составляется без явного выделения и управления параллельными участками, а адаптация к ресурсам конкретной параллельной вычислительной системы производится автоматически без участия программиста.

Параллелизм современных параллельных вычислительных систем реализован на уровне процессов и потоков, что обусловлено особенностями современных технологий, не обеспечивающих более эффективной организации параллельных вычислений на более мелких операторных примитивах, таких как отдельные машинные команды или операторы языков высокого уровня.

Процесс [110, 117, 128] – выполняемая программа, в совокупности с ее собственным адресным пространством, текущими значениями счетчика команд, регистров и переменных. Для того чтобы несколько процессов параллельно решали общую задачу, применяются механизмы передачи данных между процессами, например каналы и очереди сообщений [110, 117, 128]. Многопроцессным параллельным программам присущи следующие ошибки.

1 Передача неправильных данных или нарушения в схеме приема-передачи сообщений между процессами, возможно приводящие к потере пересылаемых данных, обусловленные ошибками в логике решения задачи,

неверной адресацией сообщений между процессами, применением механизмов с неблокирующим чтением. Искажение смысловой части сообщения возможно, если применяется механизм, не сохраняющий границы между несколькими сообщениями, например, при обмене данными через файл.

2 Тупики, возникающие при использовании механизмов с блокирующим чтением, когда процесс-приемник ожидает некое сообщение от процесса-источника до тех пор, пока оно не будет получено. Если процесс-источник по каким-либо причинам не выполняет передачу ожидаемого сообщения, процесс-приемник оказывается заблокированным на неопределенное время.

Кроме механизмов передачи сообщений в многопроцессной параллельной программе могут применяться механизмы синхронизации, например барьеры, семафоры, мьютексы [110, 117, 128]. Механизмы синхронизации позволяют останавливать один или несколько процессов до выполнения некоторого условия. С механизмами синхронизации связаны ошибки, возникающие из-за того, что условие возобновления работы оказывается невыполнимым или некорректно сформулированным, либо неверно расставлены операторы для синхронизации. Ошибки синхронизации можно подразделить на тупики, когда процесс останавливается и впоследствии не возвращается к работе, и пробуждение процесса в неподходящее время, нарушающее логику вычислений параллельной программы.

Потоки (легковесные процессы, нити) [117] – параллельные участки программы с общей памятью. Изменения, вносимые в общие данные одним потоком, будут доступны всем остальным, поэтому механизмы передачи сообщений не применяются, и ошибки, связанные с ними, не возникают. В многопоточной программе могут использоваться механизмы синхронизации, подобные или такие же, как и в многопроцессной программе. Это обуславливает возможность возникновения аналогичных ошибок, возникающих при синхронизации потоков.

Использование общей памяти может привести к возникновению гонки [117] – такому выполнению параллельной программы, когда несколько потоков осуществляют изменение общих данных одновременно, что может привести к ошибочному значению общих данных.

Программы с неявным параллелизмом [66, 122, 123] не разбивают выполняемые вычисления на отдельные процессы, связанные между собой в ходе выполнения через различные общие ресурсы. В такой программе не используются средства управления параллельными примитивами, процессы либо потоки не выделяются явно, не применяются механизмы синхронизации и передачи данных, поэтому отсутствуют и связанные с ними ошибки. Здесь представление параллелизма на программном уровне реализуется на уровне операций, а все взаимодействия между командами осуществляются по готовности данных после завершения выполняемых функций.

1.2 Классификация свойств отладчиков параллельных программ

Для классификации отладчиков параллельных программ выбраны следующие их универсальные свойства (рисунок 1.1):

- ориентация на параллельные примитивы, во многом определяющая направленность инструментария отладчиков на выявление специфических ошибок, присущих этим примитивам;
- способ обхода операторов параллельной программы;
- распределение шага отладки, вместе с вышестоящим критерием, определяющее формы обходы и представление вычислений параллельной программы разработчику;
- тип поддерживаемых контрольных точек;
- возможности использования отладочных значений, совместно с контрольными точками позволяющие управлять ходом отладки и формально задавать и проверять спецификацию к вычислениям программы;
- направление хода отладки;

- возможности визуализации, призванные повышать эффективность определения ошибок посредством приемов отображения различных аспектов выполнения программы.

Рисунок 1.1 – Критерии классификации отладчиков параллельных программ

Рассмотрим более подробно перечисленные критерии.

1.2.1 Ориентация на параллельные примитивы

Отладчики параллельных программ являются ориентированными на параллельные примитивы, применяемые в отлаживаемой программе.

1 Отладчики, ориентированные на многопроцессные параллельные программы [22, 51, 98, 102, 103, 134], сосредоточены, прежде всего, на анализе функций передачи сообщений от процесса к процессу и выявлении возможных тупиков, возникающих при работе с механизмами передачи данных.

2 Отладчики, ориентированные на многопоточные параллельные программы [22, 95, 98, 101, 103], показывают потоки программы отдельно друг от друга и отображают, какие потоки выполняются в данный момент и какие являются заблокированными, что дает пользователю представление о синхронизации потоков. Такой способ отладки используется и для многопроцессных программ. Анализом критических секций отладчики не занимаются, так как обычно критическая секция приводит к ошибке лишь при некоторых запусках программы, проверка критических секций осуществляется верификаторами, применяющими переборные алгоритмы.

3 Отладчики, ориентированные на программы с неявным параллелизмом [32, 97, 99, 100, 124, 133], подобны отладчикам последовательных программ, их дополнительные возможности призваны правильно и эффективно отображать модель вычислений, присущую такой программе, показывать группы команд, которые могут выполняться параллельно или асинхронно, выделять точки синхронизации программы. Тем самым отладка программы с неявным параллелизмом позволяет не только отследить вычисленные значения, но и дает программисту представление об уровне параллелизма написанной им программы. Для систем программирования, использующих управление по готовности данных, удобно представление программы в графическом виде, что применяется в ряде отладчиков функциональных программ [32, 97].

1.2.2 Способ обхода операторов параллельной программы

Какие бы параллельные примитивы не использовались, не определен порядок срабатывания операторов из разных параллельных участков программы. Поэтому возможна ситуация когда запущенная несколько раз параллельная программа, выдает различные результаты для одних и тех же начальных данных, что является серьезной проблемой для ее проверки и выявления ошибки. В программе с неявным параллелизмом нет явного выделения параллельных участков, но множество операторов, входные данные которых сформированы, могут выполняться параллельно или асинхронно.

Отладчики различаются по способам обхода операторов параллельной программы.

1 Путь выполнения параллельной программы выбирается автоматически [22, 32, 51, 68, 97-103, 134]. Достоинство: меньшее время, расходуемое на отладку. Недостаток: предоставляется только один случайный путь выполнения программы из множества путей. Для программы с явным параллелизмом возможна ситуация, когда в программе присутствует ошибка, но на автоматически выбранном пути она не возникает. В программе с неявным параллелизмом приведенная ситуация невозможна, так как порядок выполнения операторов из множества тех операторов, чьи входные данные известны, не влияет на вычисления.

2 Путь выполнения параллельной программы выбирается запросом к пользователю [22, 32, 51, 98, 101, 103], в тех местах, где возможно выполнение нескольких команд. Достоинство режима запроса к пользователю состоит в том, что такая схема работы позволяет пользователю в заданных пределах управлять выполнением программы, подводя ее к интересующему для проверки и изучения пути выполнения. Недостатком является большое количество запросов к пользователю, замедляющих процесс отладки.

3 Рассматриваются несколько или все пути выполнения параллельной программы. Этот пункт является умозрительным и не используется в

существующих отладчиках, что обуславливается слишком большим объемом предоставляемой пользователю информации, анализ которой затруднителен для разработчика.

1.2.3 Распределение шага отладки

Представление промежуточных результатов отладки и форма обхода программы связаны не только с выбором пути вычислений, но и с выбором шага отладки. Под шагом отладки понимается оператор или множество операторов, которые отладчик выполняет, а затем передает управление пользователю для просмотра результатов и выбора следующих действий. Отладчики используют различное распределение шага отладки (рисунок 1.2).

1 Шаг отладки выполняется в одном параллельном участке программы (процессе, потоке, множестве операторов с известными входными данными) [22, 96-98, 101, 102] (рисунок 1.2.а). Достоинство: подробное представление вычислений программы пользователю. Недостаток: на отладку программы требуется большее количество времени, чем при других организациях шага отладки.

2 Шаг отладки выполняется во всех параллельных участках программы [22, 32, 51, 96, 98, 101] (рисунок 1.2.б). Достоинства: сокращенное время отладки и симуляция параллельного выполнения программы. Кроме этого, если отладчик показывает отдельно группы процессов способных и не способных выполнить шаг отладки, то это повышает эффективность обнаружения ошибок связанных с синхронизацией процессов или потоков и с тупиками при посылке-приеме сообщений. Относительный недостаток: больший объем отладочной информации, предоставляемой на каждом шаге.

3 Шаг отладки выполняется в отмеченных пользователем параллельных участках программы [51, 44, 98, 103] (рисунок 1.2.в). Этот режим объединяет достоинства двух других режимов и дает программисту возможность управлять путем выполнения программы.

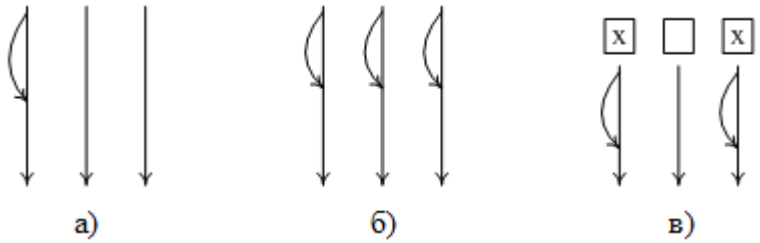


Рисунок 1.2 – Режимы распределения шага отладки по параллельным примитивам

1.2.4 Типы контрольных точек

Отладчики могут предоставлять возможность установки на операторах программы контрольных точек, изменяющих распределение шага отладки. Контрольные точки предоставляют разработчику механизм контроля и управления процессом отладки программы, отслеживания состояния на нужных участках или переменных программы. При использовании контрольных точек, поддерживающих условия пользователя, повышается шанс отследить ожидаемые логические ошибки в параллельной программе. Под условиями пользователя понимаем логические выражения, использующие значения идентификаторов программы, возможно прикрепленные к конкретным операторам программы.

1 Точка останова [32, 51, 97, 98, 101, 103, 133] определяет место в программе, до достижения которого программа выполняется отладчиком без предоставления информации пользователю и получения от него команд. Точка останова может поддерживать пользовательское условие, тогда по достижении такой точки происходит вычисление условия и в зависимости от истинности или ложности условия, выполняется какое либо действие, например приостановка отладки.

2 Точка наблюдения [98, 103] связывается с переменной и останавливает выполнение программы при каждом изменении значений этой переменной. Точка наблюдения может быть установлена как пользовательское условие или выражение, тогда приостановка вычислений будет происходить каждый раз при изменении его значения, а не отдельной переменной.

Особенности использования контрольных точек в процессах и потоках могут отличаться в различных отладчиках, так контрольная точка может устанавливаться и отслеживаться как общая для всех параллельных примитивов с общим кодом, либо применяться для отдельных процессов или потоков.

1.2.5 Возможности использования отладочных значений

Контрольные точки могут использоваться совместно с добавлением логических условий, позволяющих вычислять дополнительные выражения с подстановкой вычисленных в программе значений. В отладчиках реализованы и другие возможности использования отладочных значений.

1 Отладочные переменные [98] подразумевают, что программист во время отладки может ввести дополнительные переменные и задать выражения, изменяющие их значения в зависимости также и от переменных программы. Это позволяет отслеживать различные аспекты вычислений программы и производить дополнительные вычисления, не внося изменений в программу. Отладочные переменные и выражения дают возможность описания спецификации – набора требований к программным вычислениям, а также ее проверки для указанных начальных данных и для выбранного при отладке пути выполнения.

2 Изменение значений переменных программы пользователем во время отладки [98]. Достоинство: возможность, не начиная отладку заново, перейти к рассмотрению интересующих вычислений. Недостаток: возможное несоответствие поведения программы в процессе отладки и ее непосредственного выполнения.

3 Опции отладки частично незаконченных программ [32]. Могут быть реализованы, например, как запрос к пользователю о значении не существующей, но вызываемой в программе функции. Достоинство: отладка незаконченной программы. Недостаток: возможное несоответствие поведения программы в процессе отладки и ее непосредственного выполнения.

1.2.6 Направление хода отладки

Отладчики поддерживают различное направление хода отладки.

1 Прямой ход. Шаги отладки совпадают с естественным вычислением параллельной программы.

2 Прямой и обратный ход [97, 101]. Под обратным ходом понимают возможность на любом шаге отладки, отличном от первого, вернуться к предыдущему шагу. Это позволяет разработчику вернуться к интересующим вычислениям. В совокупности с опциями управления параллельными участками или изменения значений переменных программы, обратный ход предоставляет возможность, не начиная отладку заново, рассмотреть несколько возможных путей выполнения параллельной программы.

1.2.7 Возможности визуализации

Ряд отладчиков визуализирует графическое представление различных аспектов процесса отладки программы.

1 Визуализация блок-схемы или графа программы [32, 97]. В параллельной программе, использующей процессы или потоки, блок-схема строится для каждого параллельного примитива отдельно. Для программы с неявным параллелизмом характерно представление всей программы или отдельных функций в виде графа. Достоинства: графическое представление программы может быть использовано для улучшения восприятия самой программы и взаимодействия ее параллельных частей, для отображения вычисленных значений и связей между ними, для расстановки точек останова и логических условий на узлах схемы, для визуального выделения не только ошибок, но и пути вычисления операторов, приведшего к ошибке.

2 Визуализация передачи сообщений [22, 51, 102, 103] актуальна для многопроцессных параллельных программ. Такое графическое отображение исключительно полезно для нахождения отосланных, но не принятых сообщений

или для выделения незавершенных функций получения сообщений. Графическое представление передачи сообщений не только отображает объем взаимодействия между процессами, но и является инструментом поиска тупиков и ошибок, связанных с применением механизмов пересылки сообщений.

3 Визуализация синхронизации [44, 51, 101, 103] параллельных участков программы для программ с неявным параллелизмом заключается в визуализации графа программы. Для многопроцессных или многопоточных параллельных программ графическое отображение синхронизации может быть связано с передачей сообщений, или представлять несколько групп процессов-потоков, находящихся в таких состояниях, как: работают, ожидают, завершены. Такая визуализация помогает в поиске ошибок, связанных с тупиками или неверной остановкой-продолжением работы параллельных участков.

4 Визуализация ошибок [97, 99] состоит в графическом выделении ошибок на блок-схеме или графе программы или в ее тексте. Отладчик распознает ошибку, если результат оператора является кодом ошибки. Выделение неверно вычисленных операторов может происходить по прямому указанию пользователя или по получению от него логических условий для отдельных операторов программы. Это позволяет зафиксировать все операторы программы, на которых возникла ошибка.

5 Визуализация массивов [102, 103] отображает одномерные или двумерные массивы в графическом виде, например, выделяя цветом максимальные и минимальные элементы или определенные числа. Такая возможность может быть полезной при отладке параллельных программ, связанных с обработкой массивов.

6 Визуализация динамических структур [93, 100] представляет динамические списки или деревья в текстовом или графическом виде. Например, отображая последовательно содержимое узлов списка или создавая граф, содержащий значения узлов дерева. Визуализация динамических структур при отладке полезна для исследования работы программ, использующих такие структуры.

7 Визуализация статистических данных [51] представляет графики, таблицы и диаграммы различной сопутствующей информации о работе параллельных участков программы, например, время работы и остановки процесса, количество выделенной процессу памяти. Эта возможность не влияет на обнаружение ошибок, она позволяет изучить дополнительные особенности выполнения программы, определить эффективность и оптимизировать работу приложения.

1.3 Особенности отладки функционально-потоковых параллельных программ

Для функционально-потоковой параллельной программы характерны следующие особенности [65, 66, 68, 69].

1 Отсутствие явного (императивного) управления вычислениями. Команда, расположенная в тексте программы ниже, может быть вычислена ранее, чем команда, расположенная выше.

2 Отсутствие управления параллельными примитивами, механизмов передачи данных и механизмов синхронизации, а значит и ошибок, связанных с этими механизмами. Вместо этого используется управление по готовности данных, предполагающее, что любой оператор может быть выполнен, если известны все его входные аргументы.

3 Операторы могут быть объявлены задержанными. Их вычисление может начаться, только если произойдет раскрытие задержанного списка и все входные данные будут готовы. Если какой-либо оператор среди своих входных данных получает результат работы задержанного оператора, который не был открыт и другие операторы не могут быть выполнены, значение задержанного оператора установится в специальную константу. То есть такая ситуация не приведет к тупику, но возможно появление ошибки при дальнейшей обработке константы.

4 Результатом вычисления оператора может быть константа ошибки. Оператор, получивший ошибку как входное значение, будет ее обрабатывать и сформирует результат, в подавляющем большинстве случаев это будет снова ошибка.

5 Функционально-потоковая параллельная программа может быть представлена в виде ациклического графа, вершины которого это операторы, а дуги это связи между операторами.

Хотя модель функционально-потоковых параллельных вычислений исключает некоторые ошибки, характерные для многопроцессных и многопоточных программ, ее особенности ведут к специфическим ошибкам, в основном связанным с логикой построения программы или с неверным оперированием задержанными списками. Это требует специальных подходов к отладке программы.

Способ обхода операторов параллельной программы подразумевает метод выбора подмножества параллельных участков для выполнения шага отладки. При рассмотрении ФПП программ под параллельными участками будем понимать операторы, принадлежащие одному слою (ярусу) графа функции.

Различный порядок обхода операторов в ФПП программе не может привести к вычислению различных значений на этих операторах, как это может случиться в многопоточных и многопроцессных системах, поэтому выбор различных путей обхода операторов программы или рассмотрение всех возможных путей не способны открыть новые особенности ее исполнения. Поэтому для отладки функционально-потоковых параллельных программ наиболее подходит автоматический выбор пути выполнения программы.

Выбор шага отладки определяет набор вычисляемых операторов до передачи управления пользователю.

1 Шаг отладки выполняется в одном параллельном участке программы. Под параллельными участками будем понимать отдельные операторы из множества операторов с известными входными данными. На шаге отладке выбирается один из операторов, способных вычислить результат. Достоинства и

недостатки полностью совпадают с использованием режима в программах с явным параллелизмом.

2 Шаг отладки выполняется во всех параллельных участках программы. За один шаг отладки выполняются все операторы с известными входными значениями. Для функционально-потоковой параллельной программы, в отличие от многопроцесной, такой режим не влияет на выявление тупиков, но он дает представление пользователю об уровне параллелизма составленной им программы, показывая на каждом шаге количество операторов способных выполниться параллельно.

3 Шаг отладки выполняется в отмеченных пользователем параллельных участках программы. На каждом шаге отладки пользователю предоставляется право выбора нескольких операторов из множества готовых к выполнению операторов. Достоинство: управление путем вычисления программы. Недостатки: большой объем запросов к пользователю и как следствие большее время, требуемое для отладки, а также отсутствие зависимости между порядком вычисления готовых к исполнению операторов и генерируемыми ими данными уменьшает значимость этой возможности.

Для функционально-потоковой модели параллельных вычислений можно предложить дополнительные режимы выбора шага отладки, связанные с порядком обхода графа программы. Например, для шага отладки можно выбрать множество операторов графа образующих ветвь графа, выходящую из одной начальной вершины, и завершающую оператором, требующим входные данные от оператора из другой ветви. То есть приведенный режим будет выполнять в строгом порядке множество операторов, следующих один из другого и получающих друг от друга входные данные. Таким образом, за один шаг отладки будут выполняться несколько операторов, вычисления которых напрямую связаны между собой. Это может облегчить восприятие отладочной информации по сравнению с режимом, где шаг отладки выполняется для всех операторов с известными входными значениями, то есть операторами, вычисления которых не связаны между собой.

Контрольные точки.

1 Точка останова может быть связана с оператором функционально-потоковой параллельной программы, оператор может быть выбран как в тексте программы, так и на ее графе. С выделенным оператором также можно связать логическое условие, в зависимости от значения приостанавливающее отладку или продолжающее вычисления. В отличие от программ с явным параллелизмом, при нескольких запусках отладки с точкой останова на одном и том же операторе, возможно вычисление различного числа операторов до достижения точки останова, это связано с неопределенным порядком вычисления набора команд с известными входными значениями.

2 Точка наблюдения, останавливающая отладку, если изменилось значение связанной с ней переменной, не может быть применена в функционально-потоковой параллельной программе. Это обусловлено отсутствием переменных, вместо которых используется принцип единственного обозначения для идентификаторов и операторов программы [68]. Точка наблюдения, связанная с введенным пользователем условием, например останавливающая отладку, когда условие станет ложным, может быть применена. В этом случае необходимо учитывать, что значения идентификаторов, используемых в пользовательском условии, будут неизвестны с начала выполнения функции и до их вычисления. Точка наблюдения с условием полезна для более быстрого определения, соответствует ли вычисление программы ожидаемым результатам, чем отладка по шагам.

Контрольные точки в функционально-потоковой параллельной программе могут быть связаны с оператором или пользовательским выражением, но не с идентификатором программы.

Использование отладочных значений в функционально-потоковой модели параллельных вычислений.

1 Отладочные значения в функционально-потоковой параллельной программе являются дополнительными идентификаторами или формулами,

введенными пользователем перед началом или во время отладки и не изменяющими саму отлаживаемую программу. Использование отладочных значений позволяет визуализировать вычисления, интересные разработчику, без изменения кода или проверять условия пользователя, связанные с вычислениями, что способствует выявлению логических ошибок в программе.

2 Изменение значений переменных программы во время отладки может быть реализовано как возможность изменения результата вычисления любого оператора или значения идентификатора. Достоинства и недостатки такой опции отладки для функционально-потоковых параллельных программ аналогичны достоинствам и недостаткам последовательных и многопроцессных программ.

3 Опции отладки частично незаконченных программ позволяют пользователю ввести значение результата выполнения не существующей или незаконченной функции, а также невычисленного идентификатора.

В функционально-потоковой параллельной программе **направление хода отладки** может быть и прямым, и обратным при любом описанном режиме определения шага отладки.

Возможности визуализации функционально-потоковой параллельной программы связаны с отображением ациклических графов отдельных функций или общего графа программы, начиная с вызова выбранной функции.

1 Визуализация графа программы корректно отображает модель вычислений, их возможный порядок, точки синхронизации программы и уровень ее параллелизма. Вершины графа представляют операторы и могут хранить как программную запись оператора, так и его вычисленное значение, дуги показывают связи между операторами, отображают зависимости между выходными и входными значениями групп операторов, показывая тем самым множества операторов, порядок выполнения которых не определен заранее, и выделяя синхронизацию в программе. Дуги могут предоставлять информацию о вычисленных значениях операторов. Для ФПП программы, состоящей из нескольких функций, возможны два варианта визуализации графа: как множество не связанных между собой графов отдельных функций или общий граф всей

программы, обусловленный иерархией вызовов функций относительно стартовой функции. Последний способ визуализации дает более полное представление о степени параллелизма программы, но при наличии рекурсивных вызовов или достаточно большом числе задействованных функций является сложным для восприятия.

2 Визуализация передачи сообщений не требуется, по причине отсутствия механизмов передачи сообщений и необходимости в них.

3 Визуализация синхронизации включена в отображение графа программы и определяется связями между операторами и информационными зависимостями между функциями.

4 Визуализация ошибок реализуется как выделение цветом операторов-вершин выдавших неверные значения - это: константы ошибок, ложные пользовательские условия, приписанные к оператору, или указание разработчика выделить вершину как ошибочную.

5 Визуализация списков может быть выполнена наподобие визуализации массивов, например, как выделение цветом элементов в заданном интервале, максимальных и минимальных значений и т.п. Отличие состоит в том, что если в последовательных программах или в параллельных примитивах массив для отображения выделяется по переменной и его визуализация может измениться по мере выполнения команд, то список может определяться как входное или выходное значение указанного оператора. Выделение конкретного списка для визуализации можно не использовать, удобнее вместо этого применять параметры визуализации к любому значению оператора-вершины, выбранному для просмотра.

6 Визуализация статистических данных предоставляет на графе программы время выполнения для каждой вершины-оператора или отображает таблицы или диаграммы времени выполнения отдельных функций программы или объема используемой ими памяти.

Все возможности отладчиков из приведенной классификации могут быть применены или адаптированы для функционально-потоковой параллельной

модели вычислений, за исключением точек наблюдения, связанных с переменными, и визуализации передачи сообщений. Отличительная возможность отладки функционально-потоковых параллельных программ – это дополнительные режимы шага отладки, связанные с различными способами обхода ациклического графа программы.

Предложенные опции для отладки ФПП программ позволят повысить эффективность отладки. Различные опции позволят получить различные эффекты (таблица 1), например, такие как уменьшение времени отладки, повышение эффективности локализации логических ошибок, определение формальной спецификации программы, отображение уровня параллелизма и временных затрат на вычисления, контроль над шагом и путем обхода во время отладки, детальное рассмотрение вычислений программы.

Таблица 1 – Опции для отладки ФПП программ

Опция	Эффек-тивность	Особенности
1	2	3
Способ обхода операторов параллельной программы		
Путь выполнения выбирается автоматически	+	Повышение скорости отладки за счет уменьшения числа запросов к разработчику
Путь выполнения выбирается запросом к пользователю	+/-	Различные пути обхода не изменяют вычислений; большой объем запросов к пользователю; управление ходом отладки
Рассматриваются несколько или все пути выполнения	+/-	Различные пути обхода не изменяют вычислений; большой объем предоставляемой информации; опция полезна при наличии асинхронных команд

Продолжение таблицы 1

1	2	3
Распределение шага отладки		
Шаг отладки в одном параллельном участке	+	Детальное рассмотрение вычислений
Шаг отладки во всех параллельных участках	+	Повышение скорости отладки; отображение уровня параллелизма
Шаг отладки в отмеченных пользователем параллельных участках	+/-	Управление путем обхода программы; но различные пути обхода не ведут к различным вычислениям, если нет асинхронных команд
Шаг как ветвь операторов информационного графа программы	+	Повышение скорости отладки; недостаток: в ряде программ может совпадать с шагом отладки в одном параллельном участке
Типы контрольных точек		
Точки останова без условий	+	Управление шагом отладки; повышение скорости отладки
Точки останова с условиями	+	Задание формальной спецификации; эффективность выявления логических ошибок
Точки наблюдения без условий	+/-	По функциональности будет совпадать с точкой останова
Точки наблюдения с условиями	+	Задание формальной спецификации; выявление логических ошибок
Использование отладочных значений		
Отладочные значения	+	Дополнительные вычисления без изменения кода программы; спецификация программы

Продолжение таблицы 1

1	2	3
Изменение значений переменных программы во время отладки	+	Варьирование рассматриваемых аспектов вычислений, но возможна потеря связи между отладкой и реальным выполнением
Опции отладки частично не законченных программ	+	Отладка не завершенной программы, но возможна потеря связи между отладкой и реальным выполнением
Возможности визуализации		
граф программы	+	Повышение наглядности отладки, использование при добавлении контрольных точек и логических условий, графическое выделение ошибок
передача сообщений	-	Не требуется за отсутствием механизмов передачи сообщений
Синхронизация	+/-	Подразумевается при визуализации графа
Ошибки	+	Разметка ошибочных вычислений повышает эффективность отладки
Списков	+	Повышение эффективности поиска ошибок в задачах, связанных с обработкой списков
статистические данные	+	Исследование технических аспектов вычислений позволит корректировать код программы с целью уменьшения ее времени выполнения

1.4 Выводы

На основе анализа существующих инструментальных средств, предназначенных для отладки параллельных программ, предложена классификация функций параллельных отладчиков, включающая такие критерии как ориентация на параллельные примитивы, способ обхода параллельной программы, распределение шага отладки, виды контрольных точек, использование отладочных значений, направление хода отладки, графическое представление процесса отладки.

Проведенный анализ позволяет предложить набор функций и вариантов их реализации в инструментальной среде, обеспечивающих эффективную отладку функционально-потоковых параллельных программ.

2 Методы верификации программ

Под верификацией [1, 27, 57, 58, 59, 80, 113, 126, 127] понимают подтверждение корректности программы [3, 11, 38, 39, 43, 45, 75, 112, 118], при этом программа считается корректной, если она удовлетворяет спецификации [8, 13, 42, 79] - набору утверждений, заданному разработчиком, определяющему свойства программы.

Методы верификации подразделяются на методы доказательства теорем [4, 28, 29, 35, 37, 40, 49, 76, 84-92, 116, 129] и методы проверки моделей [46, 47, 132]. Они предоставляют различные подходы к форме спецификации программы и к приемам подтверждения корректности.

В разделе рассматриваются методы верификации программ и формы задания спецификации, выбираются методы наиболее подходящие для верификации функционально-потоковых параллельных (ФПП) программ, приводятся примеры аналитической верификации ФПП программ с помощью выбранных методов.

2.1 Верификация функционально-потоковых параллельных программ с помощью методов доказательства теорем

Особенности модели функционально-потоковых параллельных вычислений, такие как отсутствие механизмов синхронизации и передачи сообщений, независимость вычисленных данных от способа обхода параллельных команд, позволяют для верификации функционально-потоковых параллельных программ использовать методы доказательства теорем, применимые для последовательных программ. Использование этих методов будет иметь такие же недостатки и достоинства, как и для последовательных программ или программ с явным параллелизмом. Это высокая сложность автоматизации методов и возможность подтверждения корректности для начальных данных, заданных в общем виде.

Методы доказательства теорем проводят формальное доказательство корректности программы, используя последовательность операторов программы, спецификацию программы как набор требований к начальным, конечным данным и отдельным точкам выполнения программы, некоторое множество аксиом и систему вывода для формирования дополнительных утверждений. Программа считается корректной, если из полученных утверждений, следуют утверждения спецификации. Корректность программы исследуется относительно абстрактных начальных данных, в этом состоит преимущество методов доказательства теорем перед методами проверки моделей, рассматривающих вычисления программы над известными входными данными. Недостаток методов доказательств теорем состоит в их сложной автоматизации, часть таких методов предполагает только частичную автоматизацию, требуя от пользователя задания не только спецификации, но и утверждения для построения доказательства.

В основе проверки правильности программы относительно спецификации методами доказательства теорем, лежит метод индуктивных утверждений, основные положения которого подходят практически к любым программам [88]. Метод индуктивных утверждений определяет ход верификации последовательной программы без циклов. Другие методы, такие как метод индукции [76, 89], метод счетчиков [89], инварианты [35, 76, 89, 92], функции декремента [35, 76, 89], применяются совместно с методом индуктивных утверждений, расширяя его инструментарий для верификации последовательных программ с циклами и рекурсиями. При верификации параллельных программ, опирающихся на работу потоков или процессов, каждый отдельный параллельный примитив рассматривается с помощью метода индуктивных утверждений, для подтверждения корректности всей параллельной программы может дополнительно применяться метод невзаимодействующих доказательств [89] или метод выделения коммуникационно-замкнутых слоев [45].

Метод индуктивных утверждений [35, 76, 89] проводит доказательство правильности программы, относительно ее спецификации. При этом спецификация должна содержать входное и выходное утверждения. Входное

утверждение описывает все необходимые входные условия для программы и играет роль ограничителя входных данных, для которых проверяется правильность программы. Выходное утверждение описывает ожидаемый результат вычислений, для заданного множества входных данных. Также спецификация может включать в себя промежуточные утверждения, приписанные к конкретным точкам выполнения программы. Промежуточное утверждение является начальным условием для последовательности операторов расположенных после его описания, и целью для совокупности команд, предваряющей его вхождение.

Входное утверждение предполагается истинным. Затем строится дополнительное утверждение, которое выводится на основании семантики последовательности операторов, расположенных между входным и выходным (или промежуточным) утверждениями, такое утверждение называется выведенным утверждением. Вывод утверждений проводится при помощи математических методов, использующих исчисление предикатов первого или второго порядка, и правил срабатывания программных операторов. Доказательство подтверждает, что программа соответствует спецификации, если полученные утверждения удовлетворяют выходному и промежуточным утверждениям.

Форма задания спецификации и процесс подтверждения корректности подходит для ФПП программ, при этом входное утверждение будет описывать вид аргументов функции, выходное утверждение свойства возвращаемого функцией значения, промежуточные утверждения можно прикреплять к командам в тексте программы или к операторам-вершинам ее информационного графа. В качестве схемы ФПП программы для последовательного рассмотрения утверждений и операторов наиболее подходит граф программы.

Метод индукции предлагается в [76, 89] для верификации циклов. Доказательство правильности рекурсии или цикла подобно доказательству теоремы для всех положительных чисел методом индукции. Сначала подтверждается, что спецификация выполняется на первой итерации, затем что

если она выполнилась для n-й итерации, то она также должна выполниться и для n+1-й итерации. Отсюда делается вывод, что она выполняется для всех целых чисел. Если спецификация затрагивает не сам цикл, а программу его включающую, возникает задача построения гипотезы для индукции. Такая задача решается рассмотрением выведенных утверждений, полученных на нескольких начальных итерациях, из них выделяются утверждения общего вида, при этом учитываются ситуации завершения цикла или продолжения его работы. Метод счетчиков [89] подобен методу индукции, но задает алгоритм индукции более формально и вводит дополнительные переменные для фиксирования номеров итераций.

Методы являются применимыми к рекурсивным функциям в ФПП программе, рассмотрение отдельных итераций рекурсии будет производиться посредством метода индуктивных утверждений на графе. Минус методов при автоматизации – это возможная недостаточность правил для формирования гипотезы индукции, достаточной для подтверждения спецификации.

Инварианты [35, 76, 89, 92] применяются для верификации циклов. Инвариант – это набор утверждений о свойствах выполнения цикла, описывающий ситуации по завершению цикла и продолжению его итераций, задаваемый пользователем, удовлетворяющий определенным правилам [35, 76, 89, 92]. Фактически инвариант является гипотезой для индукции, заданной человеком. Доказательство с помощью инварианта исследует ряд свойств цикла [35, 76, 89, 92] и если они выполняются, то инвариант позволяет описать состояние программы по завершению цикла и далее подтвердить корректность цикла или продолжить вывод утверждений для последовательных участков.

Доказательство с помощью инварианта, подобно методу индукции, применимо к рекурсиям в ФПП программах. Но для автоматической верификации ФПП программ выбирается метод индукции, так как и в последовательных, и в ФПП программах инвариант имеет такие недостатки: высокая сложность задания инварианта для разработчика и, как и в методе индукции, возможная

недостаточность утверждений инварианта или базы правил верификатора для подтверждения корректности программы.

Функции декремента [35, 76, 89] служат для исследования свойства завершимости цикла. Функция декремента – это функция, зависящая от переменных программы, заданная разработчиком, удовлетворяющая определенным правилам [76]. Верификация исследует ряд свойств функции [76] на итерациях цикла и если они выполняются, то отсюда заключается, что цикл не бесконечен.

Такой подход применим к рекурсивным функциям в ФПП программе, тем более что автоматическое исследование свойств функции основано на методе индуктивных утверждений. Минус подхода – это высокая сложность для пользователя составления функции декремента, подходящей для подтверждения завершимости цикла.

Методы невзаимодействующих доказательств [89] и выделения коммуникационно-замкнутых слоев [45] разделяют многопроцессную программу на части без обмена данными между процессами. Применение к ФПП программам такого подхода не даст результата из-за, как правило, большого числа связей между данными параллельных операторов. Еще более важно то, что граф программы предоставляет такую схему параллельных команд, различный способ обхода которых не приведет к различию в вычисленных данных, благодаря чему для верификации ФПП программ можно адаптировать методы верификации последовательных программ.

Из рассмотренных методов доказательства теорем для реализации автоматической верификации функционально-потоковых параллельных программ в качестве базового метода выбран метод индуктивных утверждений, а для исследования рекурсивных функций метод индукции. Предпочтение методу индукции перед инвариантами и функциями декремента отдано на том основании, что последние два метода предполагают совмещение аналитической и автоматической верификации, при этом аналитическая часть достаточно трудоемка.

2.1.1 Пример верификации функционально-потоковой параллельной программы методом индуктивных утверждений

Для использования метода индуктивных утверждений необходимо знать последовательность выполнения программы. В качестве структуры функционально-потоковой параллельной программы для метода индуктивных утверждений предлагается использовать граф функционально-потоковой параллельной программы. Такой граф является ациклическим и однозначно определяет операторы-вершины и связи между ними [65, 66, 68, 70, 71]. Возможность верификации на информационном графе программы – отличительная черта функционально-потоковой модели параллельных вычислений, достоинство которой в сужении множества утверждений, рассматриваемых для построения или проверки утверждений, вытекающих из ранее сформированных. Это связано с тем, что утверждения, получаемые при рассмотрении конкретной вершины-оператора, опираются на предыдущие утверждения не от каждого ранее рассмотренного оператора программы, а от множества вершин-операторов, связанных с рассматриваемой.

Демонстрация использования информационного графа для формирования доказательства ниже рассмотрена на примере доказательства правильности ФПП программы, вычисляющей модуль числа, методом индуктивных утверждений.

Код программы на функционально-потоковом параллельном языке Пифагор [66, 68, 69].

```
Abs << funcdef P
{ ({P:-}, P): [(P,0):(<,>=) :?]:. >> return }
```

Функция Abs получает параметр P, сравнивает его с нулем, и в зависимости от результата сравнения, возвращает либо P, либо –P.

Граф функции Abs (Рисунок 2.1), кроме представления однозначной структуры функции и синхронизации операторов, наглядно показывает вычисления, которые могут происходить параллельно.

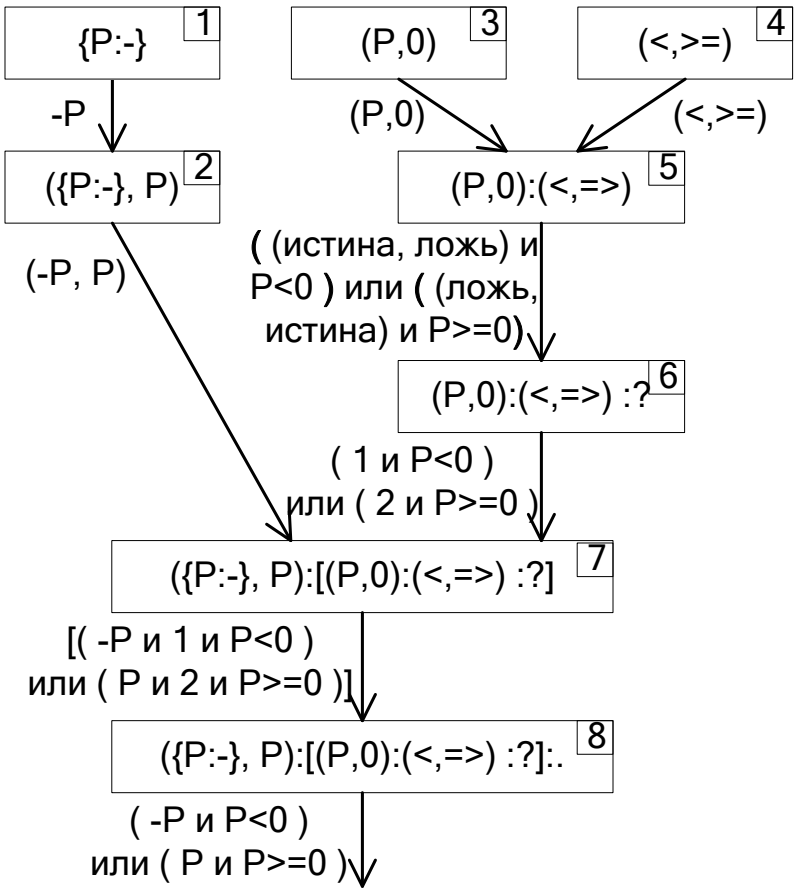


Рисунок 2.1 – Граф функции Abs с операторами и выведенными утверждениями

Для верификации функции Abs необходимо задать цель ее вычислений. Целью вычислений функции является окончательный результат, удовлетворяющий (результат = P и P>=0) или (результат = -P и P<0). Начальное утверждение: P – число. Для доказательства правильности методом индуктивных утверждений необходимо рассмотреть все операторы, согласно последовательности их вычисления, и построить выводы для каждого оператора. Для этого используем граф (рисунок 2.1).

Для вершины 1 {P:-} результат –P. Здесь вершина-оператор 1 раскрывается сразу, но возможно, а в случае задержанного рекурсивного вызова или вызова другой функции предпочтительно открытие задержанных команд по наличию требований данных от них. В примере результат оператора 1 потребуется при дальнейших вычислениях.

Для вершины 2 ({P:-},P) результат (–P, P).

Для вершины 3 (P, 0) результат (P, 0).

Для вершины 4 (<,>=) результат (<,>=).

Для вершины 5 (P,0):(<,=>) - (P<0, P=>0) результат это множество { (истина, истина), (истина, ложь), (ложь, истина), (ложь, ложь)} полученное перебором возможных значений. Где (истина, истина) соответствует выполнению (P<0 и P=>0), (истина, ложь) – (P<0 и !P=>0), (ложь, истина) – (!P<0 и P=>0), (ложь, ложь) – (!P<0 и !P=>0). Продолжая рассмотрение утверждений, получаем: (P<0 и P=>0) – не выполняется, (P<0 и !P=>0) –> (P<0 и P<0) –> P<0, (!P<0 и P=>0) –> (P>=0 и P=>0) –> P>=0, (!P<0 и !P=>0) –> (P>=0 и P<0) - не выполняется. Откуда результатом работы оператора является ((истина, ложь) и P<0) или ((ложь, истина) и P>=0).

Для вершины 6 (P,0):(<,=>) :? результат (1 и P<0) или (2 и P>=0) следует из правил работы программной функции ? и утверждения полученного в пятой вершине.

Для вершины 7 ({P:-}, P):[(P,0):(<,=>) :?] результат (-P и 1) или (P и 2), отсюда (-P и P<0) или (P и P>=0).

Полученное утверждение аналогично цели, значит, программа удовлетворяет указанной цели.

Рассмотренная функция содержит команду ветвления и не содержит циклов. Функционально-потоковая программа не может содержать циклы, но может использовать рекурсии. Так как метод индуктивных утверждений опирается на детальное рассмотрение всех операторов и всех путей выполнения программы, он является недостаточным для рассмотрения рекурсивных программ, для этого необходимы дополнительные методы, например, метод индукции.

2.1.2 Пример верификации функционально-потоковой параллельной программы методом индукции

Функционально-потоковая параллельная программа не содержит циклов, их роль играют рекурсии, проверка корректности которых может быть проведена

методом индукции. Вывод утверждений на каждой итерации метода индукции осуществляется методом индуктивных утверждений с использованием ациклического графа функционально-потоковой параллельной программы.

Рассмотрим применение метода индукции и принцип формирования гипотезы индукции на примере верификации рекурсивной ФПП программы, решающей задачу разделения множеств. Постановка задачи [78]: заданы два множества натуральных чисел S и T, сохраняя мощность множеств S и T, необходимо собрать в S наименьшие элементы множества $S \cup T$, а в T - наибольшие. Алгоритм решения: определяются максимальный элемент в множестве S и минимальный элемент в множестве T, если максимум в множестве S больше минимума в множестве T, то максимальный элемент множества S исключается из S и добавляется в множество T, а минимальный элемент множества T исключается из T и включается в множество S, иначе вычисления завершаются.

Код программы на функционально-потоковом параллельном языке.

```
Rec << funcdef x
{       s << x:1 ; max << (s,<):MaxMin ;
        t << x:2 ; min << (t,>):MaxMin ;
         [ ((max:1,min:1):[>,<=]):? ] ^ ( { block {
                s1 << s:(max:2:-):[] ; s2 << (s1:[],min:1) ;
                t1 << t:(min:2:-):[] ; t2 << (t1:[],max:1) ;
                (s2,t2):Rec >> break ; }
        }, x ):. >> return ; }
```

Функция MaxMin работает с аргументом вида (список чисел, операция < или операция >). Результатом ее работы является (максимальный элемент из списка, его порядковый номер в списке) при операции <, или (минимальный элемент из списка, его порядковый номер в списке) при операции >. Здесь при верификации программы будем считать, что функция MaxMin является корректной, то есть удовлетворяет описанным условиям. Функция Rec получает

экстремумы множеств и если максимум меньше либо равен минимуму, то рекурсия завершается, иначе производится обмен экстремумами между множествами и рекурсивный вызов над ними.

Докажем, что приведенная программа является частично корректной, пользуясь методом индукции. Свойство частичной корректности [45, 76] означает, что если программа завершится, то она завершится в состоянии, удовлетворяющем цели. Свойство тотальной корректности [45] объединяет в себе свойство частичной корректности и гарантию того, что программа завершиться. Метод индукции позволяет установить только свойство частичной корректности.

Входное утверждение программы разделения множеств – X = (S, T), где S и T – списки чисел, $|S| \geq 1$ и $|T| \geq 1$, т.е. мощности множеств больше либо равны единице. Цель вычислений ($S^{\text{конечное}}$, $T^{\text{конечное}}$), где:

1. $|T^{\text{конечное}}| = |T|$
2. $|S^{\text{конечное}}| = |S|$
3. любой элемент $T^{\text{конечное}} \geq$ любого элемента $S^{\text{конечное}}$
4. $S \cup T = S^{\text{конечное}} \cup T^{\text{конечное}}$

Анализ программы методом индукции предполагает рассмотрение нескольких итераций рекурсии. Для разделения утверждений полученных на разных итерациях, введем специальные обозначения. S, T – аргумент получаемый при первом вызове рекурсивной функции, S^1, T^1 – аргумент получаемый на второй итерации рекурсии, результат над вычисленными данными первой итерации, S^2, T^2 – аргумент получаемый на третей итерации рекурсии, результат над вычисленными данными второй итерации и т.д. Под $S^{\text{конечное}}$, $T^{\text{конечное}}$ понимается результат, вычисленный на последней итерации рекурсии, т.е. на той итерации, когда охрана (логическое выражение, управляющее вызовом рекурсии) оказалась ложной.

Для вывода утверждений строится граф функции Rec (рисунки 2.2, 2.3).

Операторы, окруженные контуром, являются задержанными, они выполняются при наличии требования данных от них другими операторами. Здесь, такое требование - это выполнение условия max>min.

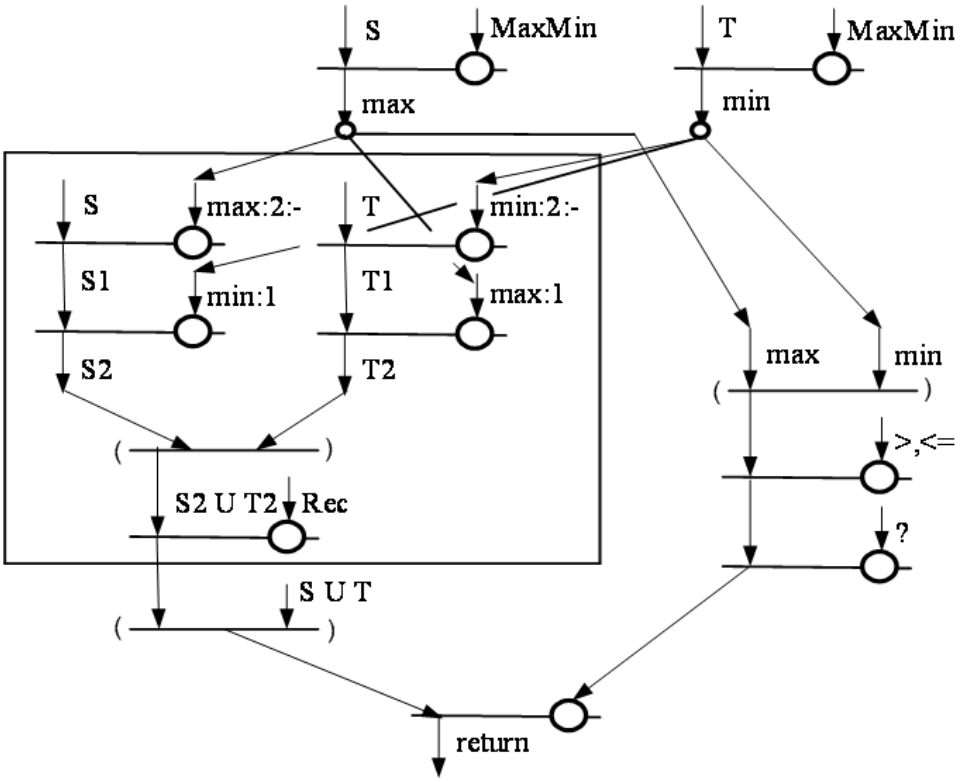


Рисунок 2.2 - Сокращенный граф функции Rec, операторы отображены графически

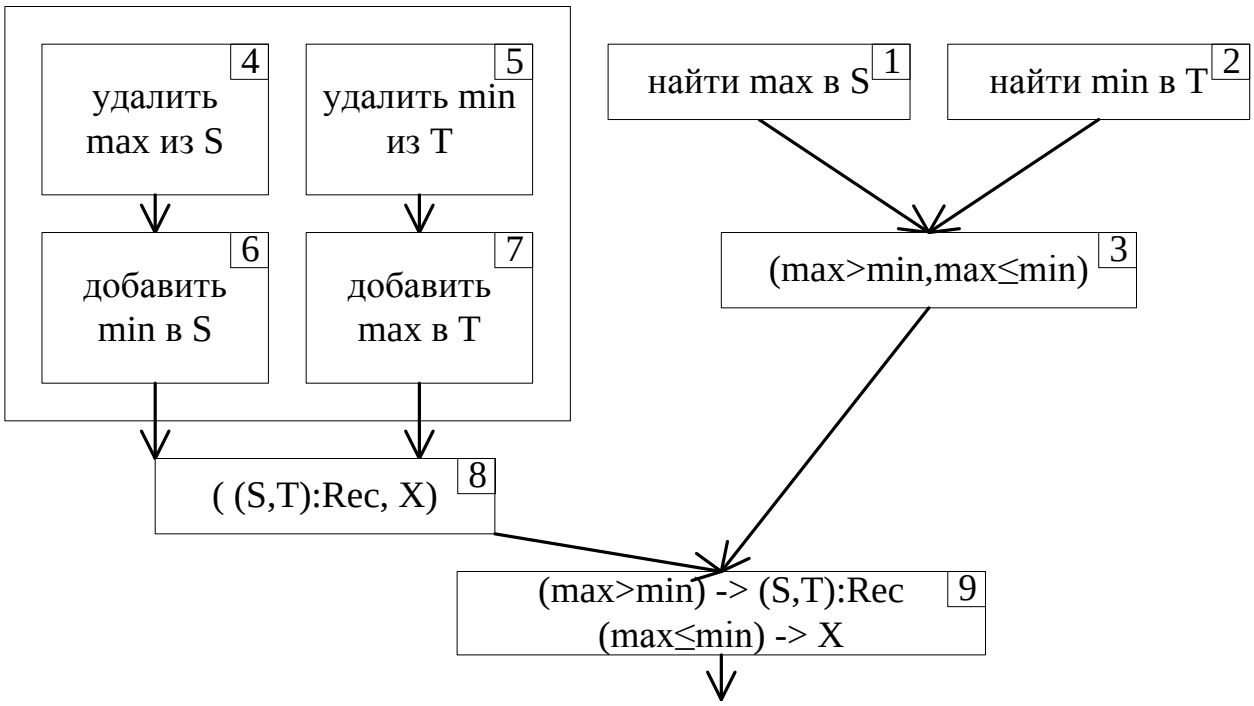


Рисунок 2.3 - Сокращенный граф функции Rec с текстовым описанием операторов

Рассмотрим первую итерацию рекурсии (рисунок 2.4) – первый шаг индукции. Результат работы оператора 1 - max ≥ любого элемента S, следует из спецификации функции MaxMin, полагаемой корректной.

Результат работы оператора 2 – min ≤ любого элемента T.

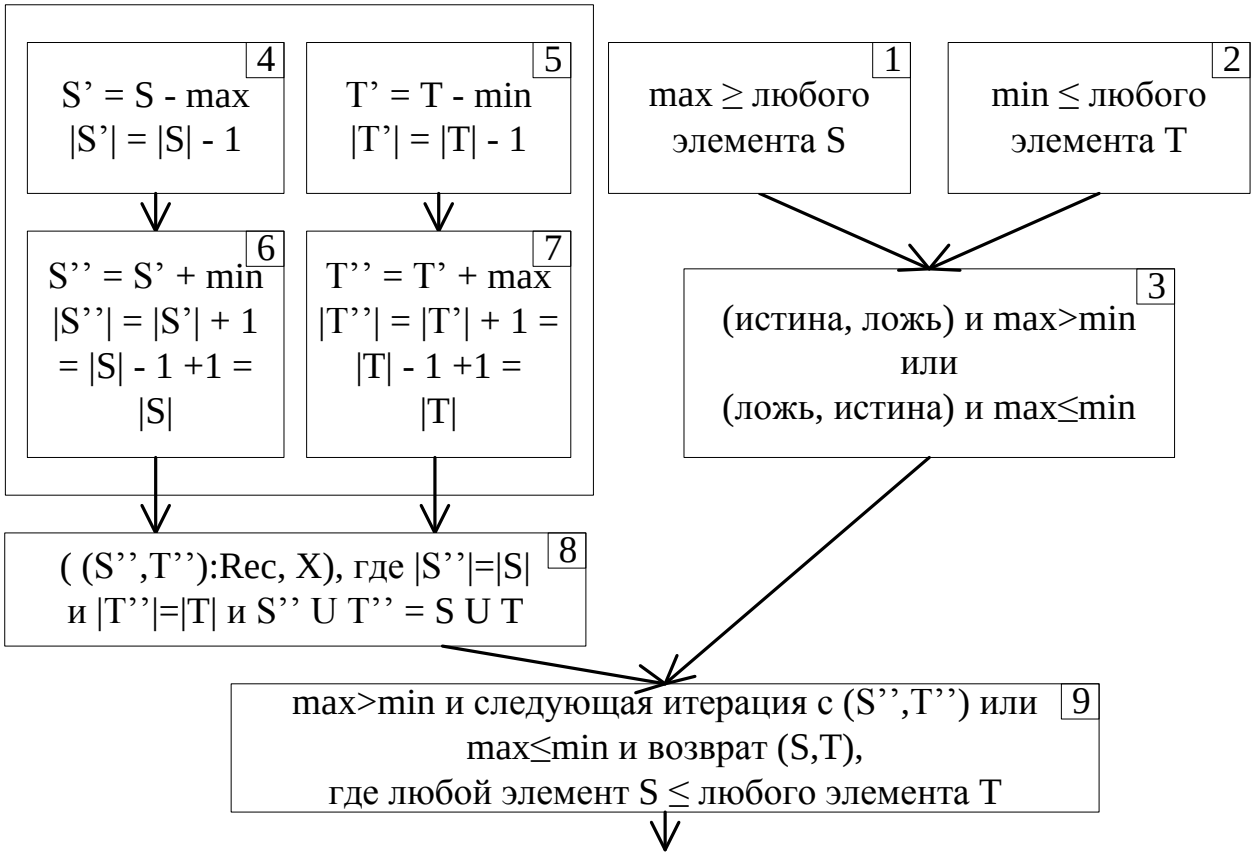


Рисунок 2.4 - Сокращенный граф функции Rec с утверждениями, выведенными на первом шаге индукции

Для оператора 3 является возможным привести все возможные варианты его выполнения – это (истина, ложь), (ложь, истина), (истина, истина), (ложь, ложь). Каждый из вариантов необходимо рассмотреть, для прояснения вопросов: не противоречат ли результаты друг другу и можно ли вывести ограничение условий для конкретной пары значений. Результат (истина, ложь) соответствует условию max>min и !(max≤min), откуда max>min и max>min, окончательное условие - max>min. Аналогично (ложь, истина) выполняется для условия max≤min. Результат (истина, истина) соответствует условию max>min и max≤min, описывающему пустое множество значений (max, min). То же самое можно сказать и о паре значений (ложь, ложь). Таким образом, постусловие оператора 3

и предусловие оператора 9 – ((истина, ложь) и max>min) или ((ложь, истина) и max≤min).

Для оператора 9 и предусловия ((ложь, истина) и max≤min) получаем в качестве результата X=(S,T), где S и T – первоначальные множества, и завершение работы рекурсии. Цель при таком завершении рекурсии выполняется. Так как множества не изменялись, то и мощность множеств и их объединение осталась прежними, т.е. пункты цели 1, 2 и 4 верны. Необходимо проверить третий пункт - любой элемент T ≥ любого элемента S. Для этого возьмем утверждения первого и второго операторов - max ≥ любого элемента S и min ≤ любого элемента T, и предусловие оператора 9 - max≤min. Получаем любой элемент S ≤ max ≤ min ≤ любой элемент T, откуда любой элемент T ≥ любого элемента S. Вывод: если рекурсия завершается на первой итерации, то цель выполняется.

Для оператора 9 и предусловия ((истина, ложь) и max>min) необходима информация из операторов 4-8. Результат действий операторов 4, 6 это множество S^1 полученное из S удалением max и добавлением min, мощность которого $|S^1|$ = |S| - 1 + 1 = |S|. Операторы 5, 7 формируют множество T^1 полученное из T удалением min и добавлением max, мощность которого $|T^1|$ = |T| - 1 + 1 = |T|. Объединение множеств S^1 и T^1 выражается как (S – max + min) ∪ (T - min + max) = S ∪ T. В операторе 9 происходит рекурсивный вызов над множествами S^1, T^1, т.е. переход к следующей итерации рекурсии.

Выходное утверждение первой итерации рекурсии это ((S^1, T^1), где $|S^1|$ = |S|, $|T^1|$ = |T|, $S^1 \cup T^1$ = S ∪ T и переход к следующей итерации) или ((S, T), где |S| = |S|, |T| = |T|, S ∪ T = S ∪ T, любой элемент S ≤ любого элемента T и завершение вычислений). Если вычисления завершаются на первой итерации, то цель выполняется.

Рассмотрим вторую итерацию рекурсии – второй шаг индукции. Вторая итерация выполняется над множествами (S^1, T^1), свойства которых были выявлены исследованием первой итерации рекурсии. Проверка второй итерации аналогична проверке первой итерации, отличие состоит в том, что на втором

прохождении рекурсии используются предикаты $|S^1| = |S|$, $|T^1| = |T|$, $S^1 \cup T^1 = S \cup T$.

Выходное утверждение второй итерации рекурсии это ((S^2, T^2), где $|S^2| = |S|$, $|T^2| = |T|$, $S^2 \cup T^2 = S \cup T$ и переход к следующей итерации) или ((S^1, T^1), где $|S^1| = |S|$, $|T^1| = |T|$, $S^1 \cup T^1 = S \cup T$, любой элемент $S^1 \leq$ любого элемента T^1 и завершение вычислений). Если вычисления завершаются на второй итерации, то цель выполняется.

Используя выходные утверждения первой и второй итерации, сформируем утверждение, которое хотим проверить (гипотезу) – для любой итерации n ((S^n, T^n), где $|S^n| = |S|$, $|T^n| = |T|$, $S^n \cup T^n = S \cup T$ и переход к следующей итерации) или ((S^{n-1}, T^{n-1}), где $|S^{n-1}| = |S|$, $|T^{n-1}| = |T|$, $S^{n-1} \cup T^{n-1} = S \cup T$, любой элемент $S^{n-1} \leq$ любого элемента T^{n-1} и завершение вычислений). То есть если рекурсия завершается, то цель выполняется.

Перейдем к третьему шагу индукции (рисунок 2.5). Будем считать утверждение, сформулированное в предыдущем абзаце, истинным для некоторой итерации n. Предположим n-ю итерацию не конечной итерацией рекурсии, т.е. получаем выходное утверждение итерации n ((S^n, T^n), где $|S^n| = |S|$, $|T^n| = |T|$, $S^n \cup T^n = S \cup T$ и переход к следующей итерации). Используя это утверждение, начальное утверждение, граф рекурсивной функции, рассмотрим n+1-ю итерацию (рисунок 2.5). Если в результате n+1-я итерация будет удовлетворять гипотезе, значит программа – частично корректна. Вывод утверждений для n+1-й итерации рекурсии аналогичен выводу для первой и второй итераций.

Результаты работы операторов 1, 2, 3 и 9 выводятся точно также как и при рассмотрении первой итерации рекурсии.

Результат действий операторов 4, 6 это множество S^{n+1} полученное из S^n удалением max и добавлением min, мощность которого $|S^{n+1}| = |S^n| - 1 + 1 = |S^n|$. Используя утверждение $|S^n| = |S|$ полученное на n-й итерации, получаем $|S^{n+1}| = |S|$.

Операторы 5, 7 формируют множество T^{n+1} полученное из T^n удалением min и добавлением max, мощность которого $|T^{n+1}| = |T^n| - 1 + 1 = |T^n|$. Используя утверждение $|T^n| = |T|$ полученное на n-й итерации, получаем $|T^{n+1}| = |T|$.

Объединение множеств S^{n+1} и T^{n+1} выражается как $(S^n - max + min) \cup (T^n - min + max) = S^n \cup T^n$. Используя утверждение $S^n \cup T^n = S \cup T$ полученное на n-й итерации, получаем $S^{n+1} \cup T^{n+1} = S \cup T$

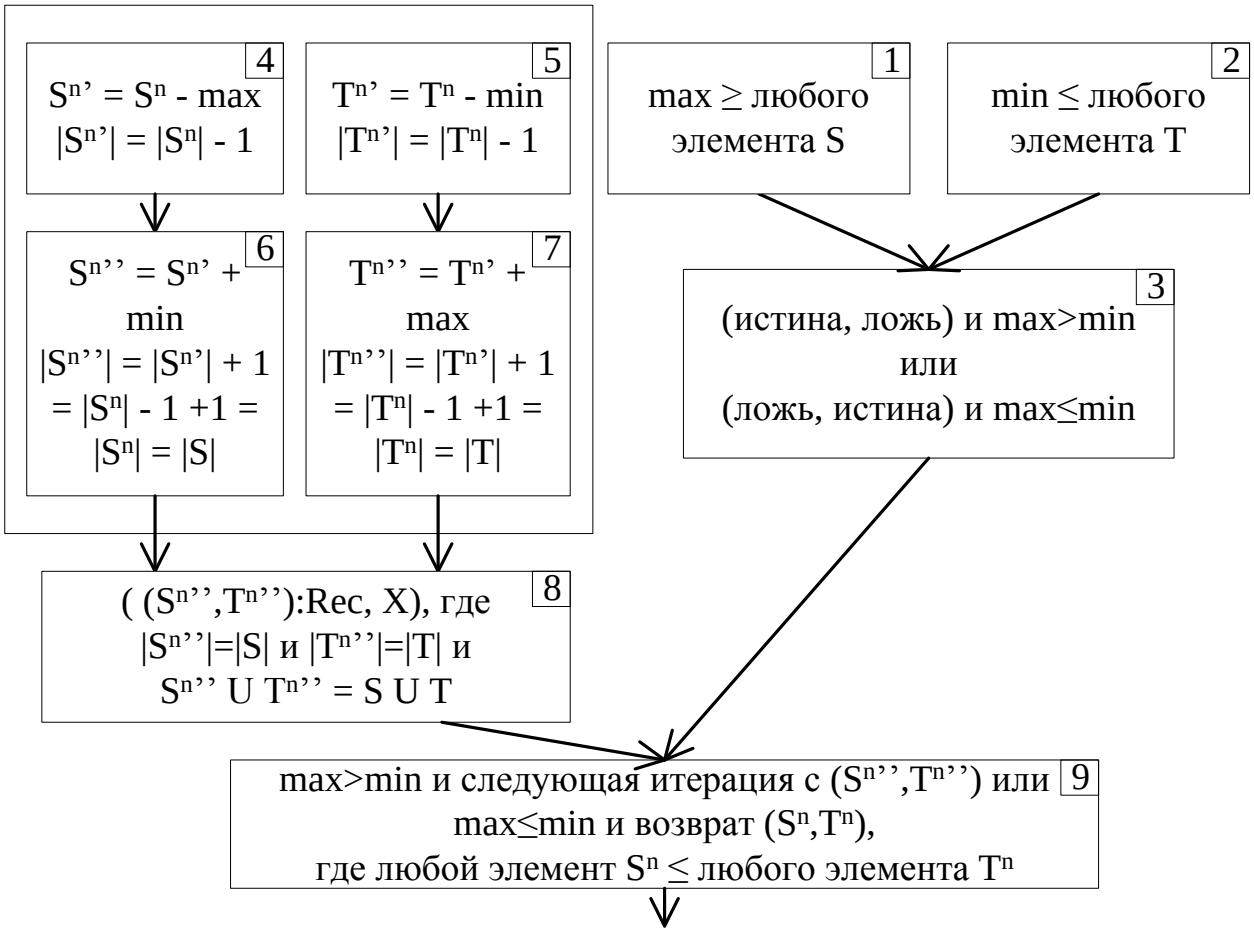


Рисунок 2.5 - Сокращенный граф функции Rec с утверждениями, выведенными на последнем шаге индукции

Из множества утверждений следует, что в завершении n+1-й итерации рекурсии выполняется ((S^{n+1}, T^{n+1}), где $|S^{n+1}| = |S|$, $|T^{n+1}| = |T|$, $S^{n+1} \cup T^{n+1} = S \cup T$ и переход к следующей итерации) или ((S^n, T^n), где $|S^n| = |S|$, $|T^n| = |T|$, $S^n \cup T^n = S \cup T$, любой элемент $S^n \leq$ любого элемента T^n и завершение вычислений).

Гипотеза индукции выполняется, следовательно, исследуемая функция является частично корректной, то есть если рекурсия завершается, то результат вычислений рекурсии полностью удовлетворяет заданной цели.

Метод индукции результативен для ФПП программ и применяется почти также как и для последовательной рекурсивной программы, за исключением учета

задержанных вычислений, что обусловлено моделью ФПП вычислений. Тогда как для многопроцессной параллельной программы разделения множеств [69] метод индукции можно применить только после выделения всех точек синхронизации, и даже после этого дополнительно необходим учет различного порядка срабатывания параллельных команд из разных процессов.

2.2 Верификация функционально-потоковых параллельных программ с помощью методов проверки моделей

Методы проверки моделей [46, 47, 132] предназначены для автоматической верификации программ. При этом подходе для программы строится модель с конечным числом состояний (например, структура Крипке [12, 46, 47]), а спецификация выражается на языке темпоральной логики [15, 47], позволяющей сформулировать особенности последовательности наступления событий. Модель программы, начальные данные в виде конкретных значений и спецификация подаются на вход программе-верификатору, которая осуществляет проверку истинности формул посредством переборных алгоритмов.

Методы проверки моделей используются для проверки последовательных, параллельных и асинхронных программ. Они удобны для автоматизации, именно такой подход к верификации использует большинство существующих программ-верификаторов. Дополнительное достоинство методов – это возможность построения контр-примера, содержащего путь вычислений программы, не соответствующий спецификации. Но методы проверки моделей не дают ответа на вопрос о том, корректна ли программа на множестве входных данных. При их использовании программа или ее модель выполняется (если это программа параллельная или асинхронная, то проверяются по возможности все варианты ее исполнения) над конкретными начальными данными, проверка формулы основывается на подстановке конкретных значений, а не на логическом выводе, как в методах доказательства теорем.

Особенности модели функционально-потоковых параллельных вычислений, такие как независимость вычислений от порядка выполнения множества операторов с известными входными значениями, отсутствие тупиков, механизмов синхронизации и передачи сообщений, ресурсных конфликтов, снижают эффективность применения методов проверки моделей для функционально-потоковых параллельных программ, в отличие от программ с явным управлением параллельными примитивами. Это связано с тем, что методы проверки моделей осуществляют перебор множества возможных путей выполнения программы и подставляют в формулы спецификации конкретные значения, вычисленные программой. Для функционально-потоковой параллельной программы также существует множество возможных путей выполнения операторов программы, но невозможна ситуация, когда несколько путей выполнения программы приводят к различным вычисленным операторами значениям. Поэтому рассмотрение множества возможных путей прохождения функционально-потоковой параллельной программы по выявленным результатам полностью совпадет с простым вычислением программы.

Применение методов проверки моделей к ФПП программам может быть полезно в таких аспектах.

1 Спецификация в терминах темпоральной логики расширяет инструментарий формулирования требований пользователя к вычислениям программы. Расстановка промежуточных утверждений, как это определено в методе индуктивных утверждений, лишь частично покрывает выразительные средства спецификации на темпоральной логике. Например, посредством последней можно задать требование о том, что идентификатор А не будет вычислен ранее идентификатора Б и подтвердить это или опровергнуть путем перебора всех возможный путей выполнения для конкретного аргумента. Исследование подобных требований в ФПП программе относится более не к проверке логики действий программы, так как изменение порядка выполнения параллельных команд не изменит результата, а например, к выявлению ситуаций, замедляющих вычисления.

2 ФПП программа с бесконечно работающим алгоритмом, описывающимся конечным числом состояний. Если число состояний велико, на основании одной отладки сложно выявить свойства программы, тогда как метод проверки модели верифицирует требования пользователя на всех различных путях.

3 ФПП программа, использующая асинхронные списки [62, 64], предоставляющие по запросу заранее неизвестный элемент списка. Такая программа может иметь различные вычисления при разных вариантах срабатывания команд, применяемых к асинхронным спискам.

Применение метода проверки моделей полезно во всех трех ситуациях, но для реализации инструментальной программной среды для верификации ФПП программ выбрано исследование последней ситуации.

Случай второго пункта отринут из-за ограниченного класса программ, удовлетворяющих условию и необходимости программирования модуля, определяющего конечное множество неодинаковых путей вычислений или модуля перевода программы в ее модель.

Спецификация в терминах темпоральной логики не реализуется, так как модуль верификации будет основываться на методе индуктивных утверждений, а предполагаемая им форма спецификации, посредством промежуточных условий и благодаря модели ФПП программ, позволяет специфицировать логику вычислений с учетом последовательности действий (как последовательности ярусов вершин-операторов) на информационном графе программы.

Метод проверки моделей для функционально-потоковой параллельной программы с асинхронными списками планируется применять следующим образом. Для каждого оператора, обращающегося к асинхронному списку, необходимо сделать перебор всех возможных вариантов его срабатывания, а для конкретного аргумента, число таких вариантов будет конечно. При обнаружении следующего подобного оператора, необходимо также перебрать все возможные его значения, но уже с учетом всех возможных значений, полученных на предыдущем операторе и так далее. При этом проверять все возможные варианты выполнения операторов с входными данными без асинхронных списков не нужно.

Такая форма метода имеет плюс – это использование самой программы вместо ее модели, описывающейся с помощью дополнительных средств. Кроме большей простоты программной реализации это убирает возможность некорректности результатов верификации из-за неверной или неполной модели программы.

Цель применения метода проверки моделей к ФПП программе с асинхронными списками – это получение ответа на вопрос, влияет ли использование асинхронного списка на результат, вычисленный программой, и предоставление вариантов выполнения программы, приводящих к различным вычисленным значениям. В таком аспекте метод предполагается к использованию без спецификации пользователя.

2.3 Инструментальные среды для верификации программ

В настоящее время разработано достаточно много программ-верификаторов, использующих методы проверки моделей для полностью автоматической верификации как последовательных, так и параллельных или асинхронных программ. Это верификаторы Spin [22, 130], Bogor [18], Cadena, GEAR, SMV, NuSMV, Bandera, VRS, VeriSoft [57, 58], RRD [21]. Часто для верификации программы, написанной на некотором языке программирования, требуется составить ее модель на входном языке верификатора и записать исследуемые требования к программе – ее спецификацию. Часть верификаторов сразу работает с программами на практикуемых языках программирования, например Bandera и RRD обрабатывают Java-программы.

Существуют автоматические верификаторы, основанные на методах доказательства теорем или совмещающие эти методы с методами проверки моделей. Это верификаторы Isabelle [19], PVS [20], Coq, ACL2, Mizar [58, 59], VCC [23]. Верификаторы доказательства теорем часто также используют собственный язык для описания программ и требований к ним. Существуют исключения, так верификатор PVS может работать с конструкциями из языков

программирования Prolog и Lisp, а VCC исследует корректность программ на языке C.

Рассмотрение автоматических систем верификации позволило выделить следующие особенности их работы.

1 Верифицируемые свойства программ:

- спецификация пользователя – определяется соответствие вычислений программы требованиям пользователя, заданным в формальном виде, чаще всего как логические условия;

- завершимость программы – исследуется вопрос о том, является ли программа бесконечной или завершается, это затрагивает циклы, механизмы синхронизации и обмена сообщениями, в основном такой сервис предоставляют верификаторы на методах проверки моделей;

- превышение границ типов – отслеживаются все операции над переменными и отмечаются команды, способные привести к переполнению типов данных переменных.

2 Описание входных данных программы:

- конкретные значения – начальные данные задаются известными значениями переменных, такой подход реализуется в верификаторах на методах проверки моделей;

- конкретные или обобщенные значения – входные данные описываются в виде некоторого множества, например как натуральные числа, числа из заданного интервала, или с помощью указания типов данных; верификация над обобщенным множеством входных значений – отличительная черта верификаторов на методах доказательства теорем.

3 Участие пользователя в процессе верификации:

- спецификация программы – предполагает формальную запись пользователем требований к вычислениям программы, без спецификации может проверяться свойство завершения программы и переполнение типов;

- утверждения для построения доказательства – обязательно задаются пользователем в ряде методов доказательства теорем (например, инварианты и

функции декремента), ряд верификаторов позволяет задать множество аксиом, то есть утверждений, расширяющих базу знаний верификатора об особенностях работы конкретной программы, призванных помочь в процессе автоматического вывода утверждений;

- выбор метода или управление ходом метода – пользователь назначает метод для исследования программы из множества известных верификатору или управляет особенностями его работы, например верификатор Isabelle [11] позволяет разработчику выполнить желаемое число итераций индукции; управление процессом выполнения метода верификации часто обусловлено тем, что метод не дает результата при полностью автоматическом запуске.

4 Формы спецификации:

- утверждения темпоральной логики – применяются в верификаторах методов проверки моделей и посылаются верификатору отдельно от программы, они позволяют описать свойства программы, связанные с последовательностью наступления событий;

- входное (не обязательное) и выходное утверждения – могут задаваться как в тексте программы, так и отдельно, первое задает вид аргумента функции, с которой начинается выполнение, последнее ожидаемые разработчиком свойства вычисленного результата;

- входное, выходное и промежуточные утверждения – последние всегда связаны с выделенными участками программы;

- теория с аксиомами и леммами – пользователь задает знания верификатору о свойствах программы (аксиомы) и формулирует утверждения (леммы) истинность которых должен установить верификатор, такая форма спецификации задается отдельно от кода программы, но задействует идентификаторы программы и поддерживается в верификаторах на методах доказательства теорем.

В общем случае спецификация может задаваться как в коде программы, так и отдельно. Первый вариант плох тем, что при подаче кода программы стандартному компилятору придется находить и удалять либо комментировать элементы спецификации.

На текущий момент выбора методов для дальнейшей реализации автоматической верификации ФПП программ наиболее важный аспект относится к форме задания спецификации. Выбранный способ применения метода проверки моделей не требует спецификации пользователя. Выделенные методы доказательства теорем изначально сформулированы для спецификации в виде входного, промежуточных и выходного утверждений. Несмотря на достоинства спецификации в виде теории, такие как отсутствие связывания с кодом программы и определение дополнительной базы знаний верификатора, для реализации выделяется спецификация через утверждения на операторах программы. Вышеупомянутый недостаток такого подхода будет нейтрализован тем, что утверждения планируются для задания на графе программы и является возможным запоминать их отдельно от кода программы. Кроме этого, визуализация информационных графов ФПП программ предполагается для отладки и верификации, а утверждения на вершинах-операторах графа позволят наглядно выделять корректные и некорректные вычисления, что способствует повышению эффективности локализации логических ошибок разработчиком.

2.4 Методы для верификации функционально-потоковых параллельных программ

Изучение ряда методов доказательства теорем и общей идеи метода проверки модели позволило выделить для реализации автоматической верификации ФПП программ следующие методы:

- метод индуктивных утверждений, адаптируемый к ФПП программам применением на информационных графах, позволяющих сужать множество утверждений при построении выводов;

- метод индукции для рекурсивных ФПП программ, применяемый на основе метода индуктивных утверждений, остальные аспекты метода, благодаря специфике ФПП вычислений, остаются аналогичны аспектам его применения к последовательным алгоритмам;

- метод проверки модели для ФПП программ с асинхронными списками, реализация которого допустит использование не модели, а самой ФПП программы и не потребует спецификации пользователя.

Методы инвариантов и функций декремента не взяты для последующей реализации, так как частично предполагают аналитическую верификацию. Набор методов исследования корректности многопроцессных параллельных программ также отвергнут, потому что при переходе к ФПП модели, возможен вывод утверждений по информационному графу, без необходимости выделения участков процессов с коммуникацией и без, а также варьирования вариантов исполнения операторов из разных параллельных примитивов.

Для спецификации ФПП программы выбрана форма спецификации, определяемая методом индуктивных утверждений, где обязательны входное и выходное утверждение, а промежуточные утверждения не обязательны. Последние, применительно к ФПП программам, предлагается приписывать к операторам-вершинам графа. Выбор формы спецификации обусловлен тем, что основной модуль верификации будет основан на методе индуктивных утверждений с возможностями визуализации ошибочных вычислений на графе.

2.5 Выводы

Проведен анализ методов верификации программ и форм задания спецификаций. Выделены методы, реализация которых целесообразна при разработке функционально-потоковых параллельных программ.

Показано, что модель функционально-потоковых параллельных вычислений позволяет избежать ряда ошибок, связанных с тупиками, конфликтами при передаче сообщений, ресурсными конфликтами. Благодаря этому, а также ряду других особенностей для функционально-потоковой параллельной программы могут быть применены методы верификации, используемые для последовательных программ.

Приведены примеры применения метода индуктивных утверждений и метода индукции для функционально-потоковой модели параллельных вычислений.

Сформулированы условия использования метода проверки моделей для функционально-потоковых параллельных программ.

3 Методы отладки функционально-потоковых параллельных программ

Методы отладки ФПП программ позволяют использовать ряд специфических приемов, определяемых особенностью модели вычислений, в которой отсутствует понятие переменной, выполняемые функции связаны напрямую, а данные имеют динамическую типизацию. Отладка ФПП программ может быть реализована подобно отладке последовательных программ, но при таком подходе возникает задача удобного представления процесса отладки пользователю, обусловленная не обязательностью соответствия между порядком выполнения команд и порядком их записи в коде программы.

Отображение особенностей параллелизма функционально-потоковой параллельной программы базируется на использовании ее информационного графа. Кроме выделения параллельных операторов, граф применяется и для других подходов к формированию множества операторов-вершин для шага отладки или пути обхода программы. Информационный граф используется для визуализации процесса отладки и добавления отладочных формул (условий) к вершинам-операторам программы. В дополнении к визуализации информационных графов предлагаются и текстовые способы представления процесса отладки пользователю.

3.1 Режимы отладки функционально-потоковых параллельных программ

Модель вычислений определяет функционально-потоковую параллельную программу как информационный граф, в котором вершины являются операторами, а дуги определяют связи между ними. Параллельное выполнение основано на том, что одновременно выполняться могут только те операторы, между которыми нет информационных связей. Чтобы оператор мог быть вычислен, необходимо выполнение всех тех предшественников, которые связаны с ним. Для отладки функционально-потоковой параллельной программы предлагаются четыре режима [34, 120, 121], использующие информационные

графы программы, это режимы пошаговой отладки, отладки слоев, отладки ветвей и проверки формул.

3.1.1 Режим пошаговой отладки

Режим пошаговой отладки функционально-потоковой параллельной программы подобен режиму отладки последовательной программы. За один шаг отладки выполняется один программный оператор из множества операторов программы, все входные данные которых известны. Возможность использования режима пошаговой отладки для параллельной программы обусловлена спецификой языка программирования. Хотя при исполнении программы может быть реализован другой путь обхода графа, чем при пошаговой отладке, эта ситуация не приведет к возникновению конфликтов, таких, как появление невычисленных вершин, бесконечное ожидание вычисления предыдущего результата, возникновение некорректных данных. Например, если выполняется множество независимых операторов, то неважно, в каком порядке они были вычислены. Процесс отладки все равно показывает верный результат для каждого оператора и позволяет выявить ошибку. Если последовательность выполнения операторов идет в глубину графа, каждая отдельная ветвь может быть выполнена отладчиком только в определенном порядке, и различные ветви, не обладающие общим путем, не могут содержать взаимосвязанные операторы. Благодаря этому, переход пошагового отладчика от выполнения одной ветви графа программы к другой не приводит к противоречиям с непосредственным выполнением программы, на каком бы уровне вложенности функций он не произошел.

3.1.2 Режим отладки слоев

Режим отладки слоев (ярусов) предполагает, что на первом шаге отладки выполняется первый слой, т.е. все те операторы, которым требуются только входные параметры и константы (рисунок 3.1). После того, как эти операторы

окажутся вычисленными, в графе программы соответственные им вершины помечаются как выполненные, и на следующем шаге для вычисления выбираются все те операторы-вершины, которые имеют на своем входе вычисленные данные, таким образом формируется следующий слой.

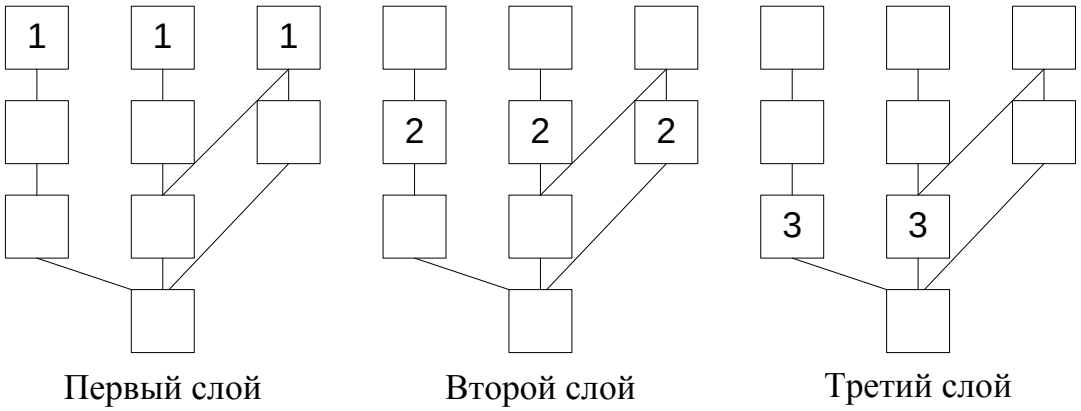


Рисунок 3.1 - Пример послойного выполнения операторов функционально-потоковой параллельной программы

В режиме отладки слоев за один шаг выполняются несколько операторов, и здесь возникает вопрос о поведении системы при возникновении ошибки на одном или нескольких операторах. Такая ситуация не является критической вследствие того, что операторы одного слоя независимы между собой. Но группа операторов в следующем слое получает ошибку как входной параметр. Если в функционально-потоковой программе некоторый программный оператор получает на входе ошибку, то он все равно производит над ней необходимые действия. Чаще всего в этом случае формируются выходные данные, также содержащие ошибку, и такие выходные данные обрабатываются последующими операторами. В процессе отладки можно рассмотреть все последствия распространения возникшей ошибки на графе программы.

Отладка по слоям предоставляет пользователю возможность наглядно увидеть количество слоев программы и количество операторов в каждом слое, то есть фактически оценить уровень параллелизма написанной программы. Такая схема выполнения наиболее точно показывает работу функционально-потоковой модели вычислений.

Описанный режим отладки обладает следующей спецификой. Если среди совокупности операторов существуют вызовы функций, то такие вызовы выполнятся сразу в полном объеме, и лишь по завершении вычисления всего слоя будет произведена отладка вызванной функции. Это не разрушает целостности восприятия слоев, но затрудняет поиск ошибки при наличии бесконечного рекурсивного вызова функции.

3.1.3 Режим отладки ветвей

Режим отладки ветвей (рисунок 3.2) предполагает, что на каждом шаге отладки для выполнения выбираются все операторы независимой ветви информационного графа.

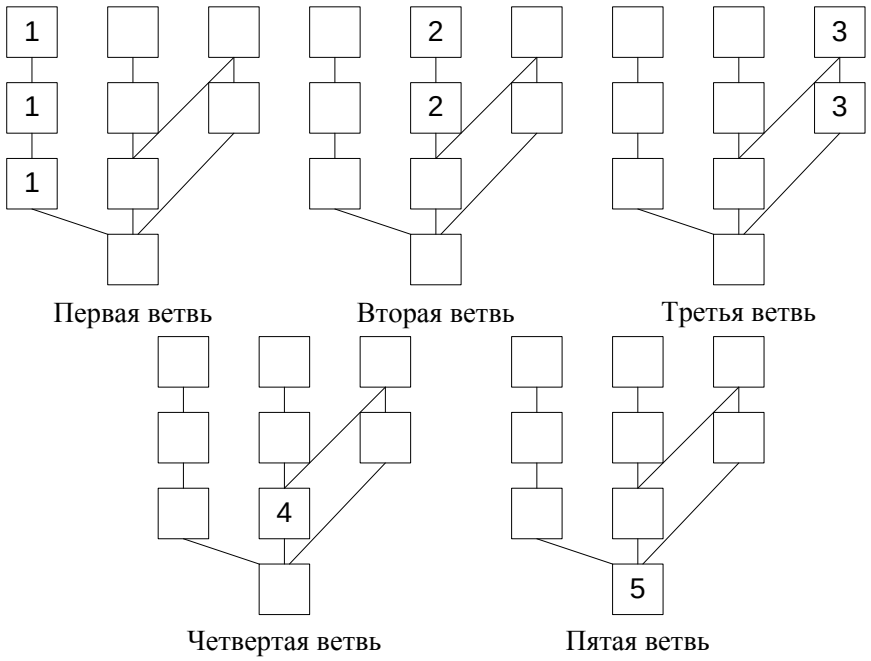


Рисунок 3.2 - Использование ветвей при отладке операторов функционально-потоковой параллельной программы

Такие операторы всегда выполняются в строгой последовательности, заданной графом программы. Это позволяет выделять в параллельной программе операторы, которые могут быть выполнены только один за другим, но при этом не нуждаются в получении каких либо других входных значений из других вершин.

Режим отладки ветвей выполняет односвязную последовательность команд за один шаг, что в определенных случаях может сократить время проверки программы.

3.1.4 Режим проверки формул

Режим проверки формул (рисунок 3.3) позволяет пользователю закрепить за произвольными узлами-операторами собственные утверждения (являющиеся выражениями на языке программирования), использующие в качестве значений аргумент функции или значения, вычисленные любым узлом-оператором графа. Если отлаживаемая функция не является рекурсивной, отладка выполняется за один шаг, на котором вычисляются все узлы графа и все дополнительные утверждения.

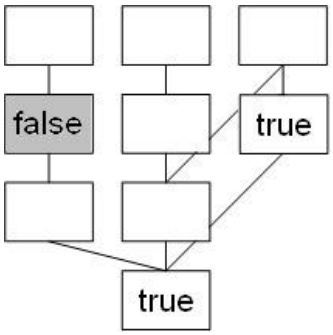


Рисунок 3.3 - Разметка вершин при отладке в режиме проверки формул

Вершины графа, содержащие утверждения, помечаются как истинные, если выражения, введенные пользователем, возвращают истину, или как ложные, если хотя бы одно из выражений не равно истине. Для каждой такой вершины вместе со значением соответствующего оператора программы можно увидеть вычисленные значения утверждений. Пользовательская формула может иметь не только логическое значение, но и любое другое: целочисленное, вещественное, строковое, но в этом случае узел графа будет помечен как ложный. Если отлаживаемая функция является рекурсивной, число шагов отладки совпадает с числом итераций рекурсивной функции, то есть пользовательские формулы повторно проверяются на каждой итерации.

Выражения, записанные пользователем, задают требования к различным частям программы. Режим проверки формул позволяет определить, соответствует ли выполнение программы требованиям пользователя при конкретных начальных данных, или вычислить интересные для пользователя выражения, не внося изменений в саму программу.

3.2 Инструментальная поддержка методов отладки

Представленные режимы отладки реализованы в среде разработки программ на функционально-потоковом параллельном языке Пифагор. Каждый из режимов обеспечивает использование графического и текстового представления программы. Графическое представление может быть отключено пользователем в режимах пошаговой отладки, отладки слоев, отладки ветвей и является обязательным для режима проверки формул.

Графическое представление программы задается в виде информационных графов всех ее функций. Перед началом вычисления любой из функций строиться ее информационный граф. При выполнении оператора внутри функции осуществляется разметка соответствующей ему вершины графа. Каждая вершина содержит доступную пользователю информацию об операторе и результате его вычисления.

Текстовое представление процесса отладки оперирует списком функций или сверткой функций. Список функций показывает историю шагов отладки, сохраняя текст выполняемой функции на каждом шаге. Свертка функции отображает этот же процесс более компактно, запоминая только предыдущее и вычисленное состояние функции.

Перед отладкой программы, вызывается диалог выбора режима отладки (рисунок 3.4), в котором разработчик определяет режим отладки, способ отображения процесса отладки и ряд других опций. «Функция для запуска» - это функция, которая будет запущена для отладки, она выбирается из списка, который формируется автоматически. «Аргумент» - входной параметр для

запускаемой функции, он задается пользователем. Функции, не имеющие входного параметра, помечаются в списке функций восклицательными знаками.

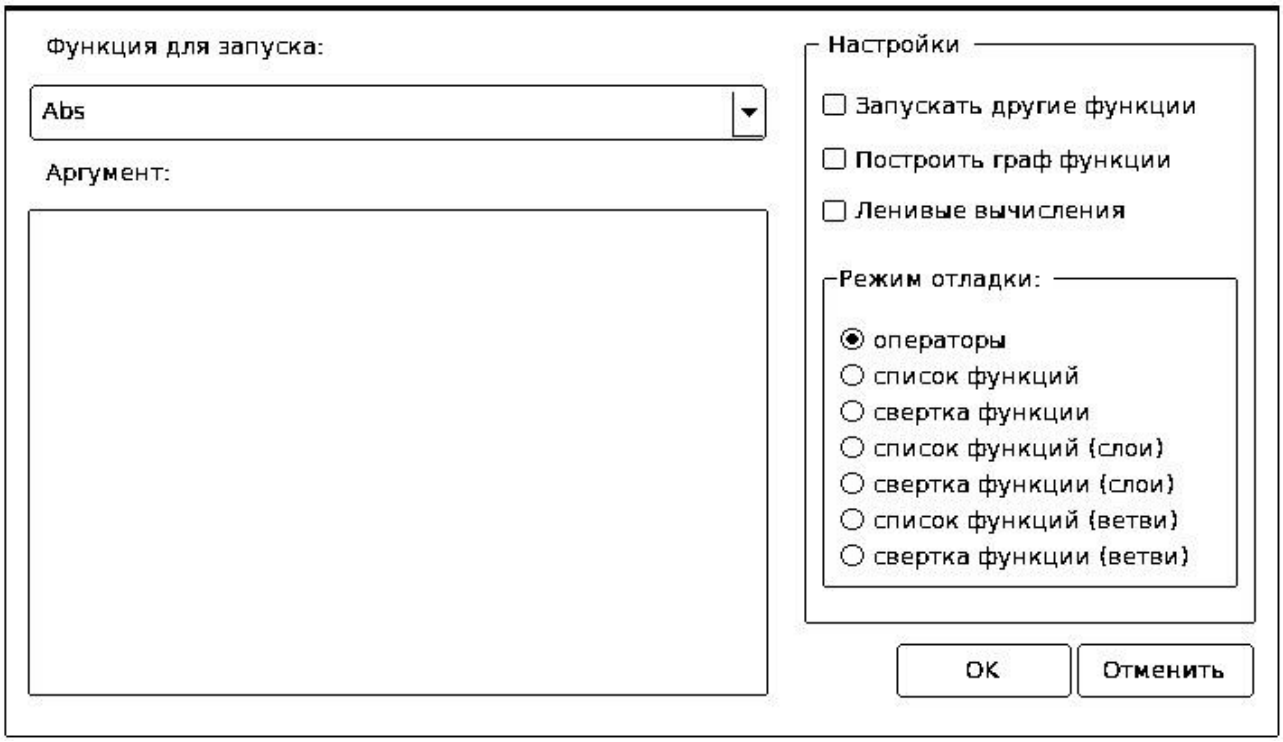


Рисунок 3.4 - Диалог выбора режима отладки

Если флаг «Запускать другие функции» отключен, то другие функции не будут детально рассматриваться во время отладки, пользователю будет предоставлен результат вычисления функции, без отображения вычислений ее отдельных операторов, это дает возможность сосредоточиться на выполнении выбранной функции, не переключая внимание на просмотр неинтересных для отладки вычислений. Если флаг включен, отладка проходит полный путь, начиная с запуска, выбранной пользователем функции. Флаг «Построить граф функции» позволяет по желанию пользователя построить граф функции. Тогда во время отладки программы будет доступен как этот граф, так и текстовое отображение процесса отладки. Флаг «Ленивые вычисления» позволяет выполнять задержанные операторы как обычные (без задержки) или вычислять их только в случае необходимости получения их значений для дальнейшей отладки функции.

Режимы «операторы», «список функции», «свертка функции» - это различные способы отображения пошаговой отладки, «список функции (слои)», «свертка функции (слои)» - различные способы отображения отладки слоев. И

«список функции (ветви)», «свертка функции (ветви)» - это различные способы отображения отладки ветвей.

Режим проверки формул запускается отдельно и предоставляет пользователю выбрать функцию и ввести ее аргумент, выбор остальных возможностей закрыт для пользователя. Режим проверки формул всегда строит граф функции и отключает запуск других функций, текстовое представление отладки совпадает с режимом «список функций».

Режим пошаговой отладки имеет один дополнительный способ текстового отображения – «операторы» по сравнению с режимами отладки ветвей и отладки слоев. Данный способ подобен большинству стандартных способов представления процесса отладки для последовательных программ. Существует текстовое поле, содержащее всю программу, по которому перемещается цветовой указатель, показывающий, какой оператор выполняется на данном шаге. Также в главном окне располагается дополнительное текстовое поле, отображающее список всех вычисленных операторов, в виде:

- исходной строки оператора;
- строки оператора после подстановки в нее всех значений, вычисленных на предыдущих шагах;
- вычисленного значения оператора.

Оператор программы, вычисляемый на текущем шаге, ставится в конец списка.

Представленный способ отображения результатов отладки отличается от методов визуализации, применяемых в последовательных программах. Если для последовательной программы операторы выполняются строго в том порядке, в каком они записаны в тексте программы, то для параллельной программы такая ситуация, скорее всего, является исключением. В функционально-потоковых языках последовательность вычисления операторов определяется не их расположением в тексте программы, а информационными зависимостями между функциями и операторами, определяемыми при построении информационного графа. Таким образом, оператор, расположенный в некоторой строке исходного

текста программы, может быть вычислен раньше операторов, стоящих в предшествующих строках. Подобная ситуация не противоречит модели вычислений потоковых параллельных программ, но является не лучшей для восприятия процесса отладки пользователем. В связи с этим были разработаны такие способы текстового представления отладки как создание списка функций и свертка функций.

Создание списка функций предполагает, что на первом шаге отладки текстовое поле дополняется телом функции, запущенной для отладки, в котором все операторы, вычисляемые на данном шаге, выделяются цветом. После получения вычисленных значений в поле текста снова дублируется тело функции, но уже с подставленными в него полученными на этом шаге отладки значениями, которые выделены другим цветом. При переходе к следующему шагу отладки поле текста снова дополняется функцией, в которую подставляются вычисленные значения, причем функция содержит и все предыдущие полученные значения. Таким образом, текстовое поле содержит список функций, где каждая последующая функция содержит те операторы, которые не были выполнены на предыдущих шагах, и также содержит все вычисленные данные.

Свертка функции отличается от списка функций только формой отображения информации, получаемой при выполнении шагов отладки. Если была выбрана свертка функции, то на экране отображается исходное тело функции, которое не меняется при отладке, а используется для выделения тех операторов, которые выполняются на данном этапе отладки. Кроме этого, отображается и текущее состояние функции, в котором записаны все предыдущие вычисления, и также добавлены данные, полученные на текущем шаге, последние выделяются цветом. Таким образом, начиная с некоторого шага отладки, исследуемая функция начинает «свертываться». На момент возврата результата вычислений функции, тело функции содержит только записи вида: значение >> идентификатор.

Оба представленных способа текстового отображения процесса отладки имеют как достоинства, так и недостатки. Свертка функции отображает процесс

отладки более компактно и наглядно, чем список функций, но при этом не позволяет просмотреть историю всех изменений и переходов.

Графическое отображение процесса отладки выполнено посредством визуализации программного графа отлаживаемой функции. По желанию разработчика оно может использоваться совместно с текстовым представлением. В программном графе еще не вычисленные вершины выделяются другим цветом, чем вычисленные. По нажатию на любую вершину графа появляется информация об исходной строке соответствующего этой вершине оператора и вычисленном значении оператора, если оно существует. Каждая вершина отображает тип соответственного ей оператора, посредством специального символа. Граф функции всегда располагает операторы-вершины одного слоя на одном уровне, независимо от того, какой режим отладки был выбран.

С формами представления режимов отладки на уровне интерфейса разработанной среды можно подробно ознакомиться в приложении А.

3.3 Пример отладки в послойном режиме

Пример послойной отладки функции, вычисляющей сумму квадратов двух чисел:

summa << funcdef x

{ ((x:1,x:1):*, (x:2,x:2):*):+ >> return; }

Пусть, в качестве аргумента функции используется список **x = (4, -2).**

Тогда, при выполнении операторов нулевого слоя произойдет выделение чисел, которые являются аргументами двух операций умножения (рисунок 3.5).

То есть, выполнятся действия в слое 0:

- 1-й оператор **x:1** преобразуется в подстановку **(4,-2):1**, и даст **4**;
- 2-й оператор **x:1** преобразуется в подстановку **(4,-2):1**, и даст **4**;
- 3-й оператор **x:2** преобразуется в подстановку **(4,-2):2**, и даст **-2**;
- 4-й оператор **x:2** преобразуется в подстановку **(4,-2):2**, и даст **-2**.

В окне отладчика исходная функция «свернется» до следующего вида:

summa << funcdef (4,-2) { ((*4* ,*4*):*, (*-2* ,*-2*):*):+ >> return;}

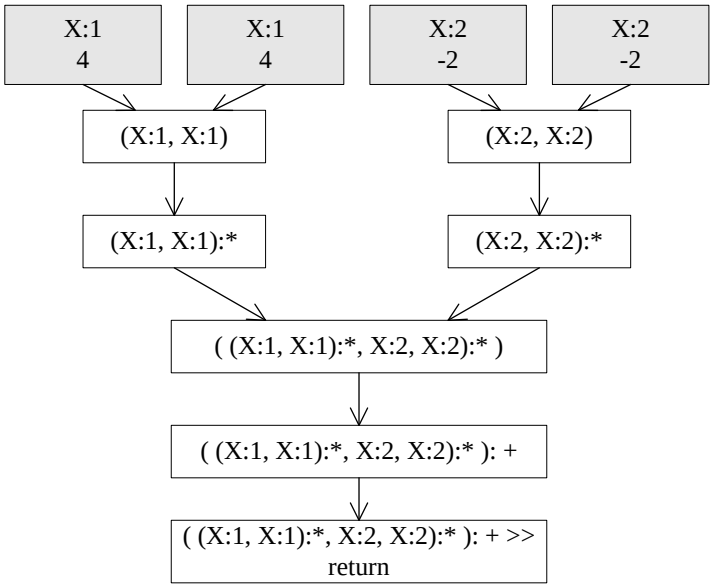


Рисунок 3.5 - Первый шаг отладки

Выполнение операторов первого слоя на втором шаге отладки (рисунок 3.6) будет связано со следующими действиями:

- оператор **(x:1,x:1)** преобразуется в **(4 ,4)**;

  оператор **(x:2,x:2)** преобразуется в **(-2 ,-2)**.

После этого шага в отлаживаемой функции произойдет выделение цветом сформированных списков (что отображено ниже курсивом с подчеркиванием):

summa << funcdef (4,-2) { (*(4 ,4)*:*, *(-2 ,-2)*:*):+ >> return;}

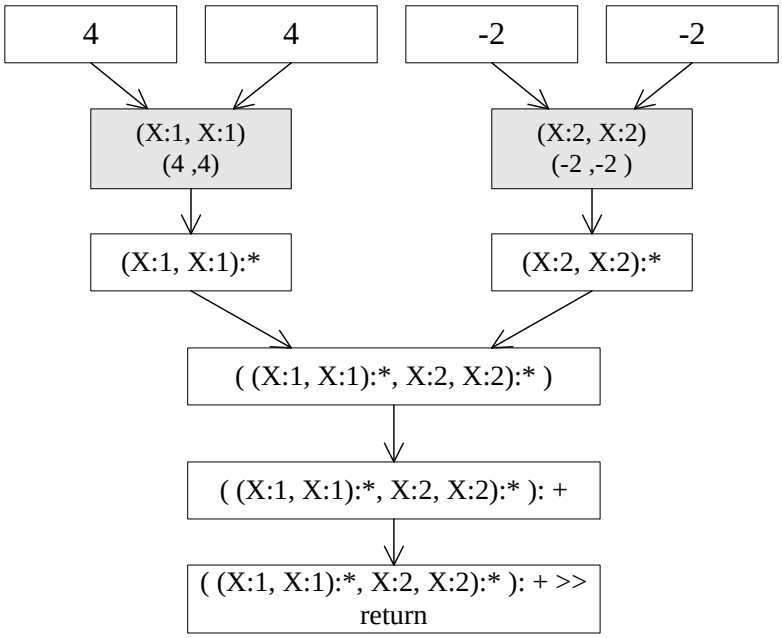


Рисунок 3.6 - Второй шаг отладки

Следующий слой связан с выполнением двух операций умножения:

- оператор **(x:1,x:1):*** реализует **(4, 4):*** и дает **16**;
- оператор **(x:2,x:2):*** реализует **(-2, -2):*** и дает **3.**

На третьем шаге (рисунок 3.7) в отлаживаемой функции произойдет выделение цветом сформированных списков (что отображено ниже курсивом с подчеркиванием): **summa << funcdef (4,-2) { (*16*, *4*):+ >> return;}**

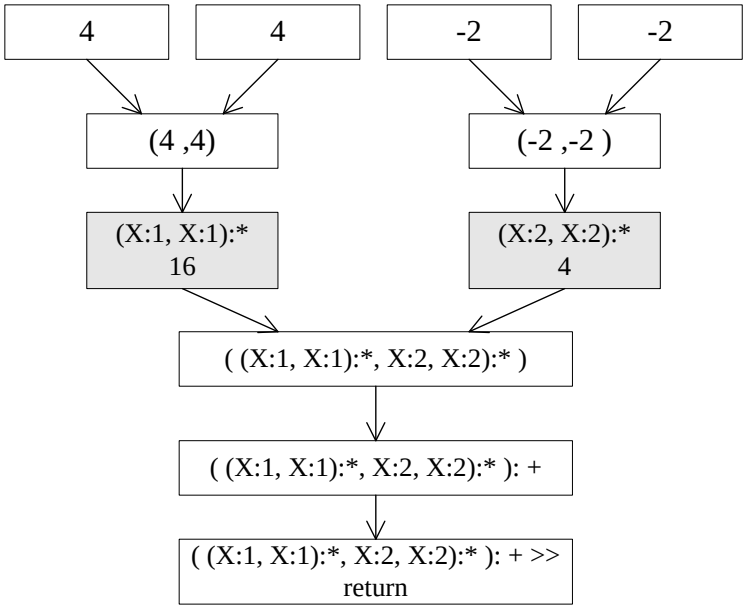


Рисунок 3.7 - Третий шаг отладки

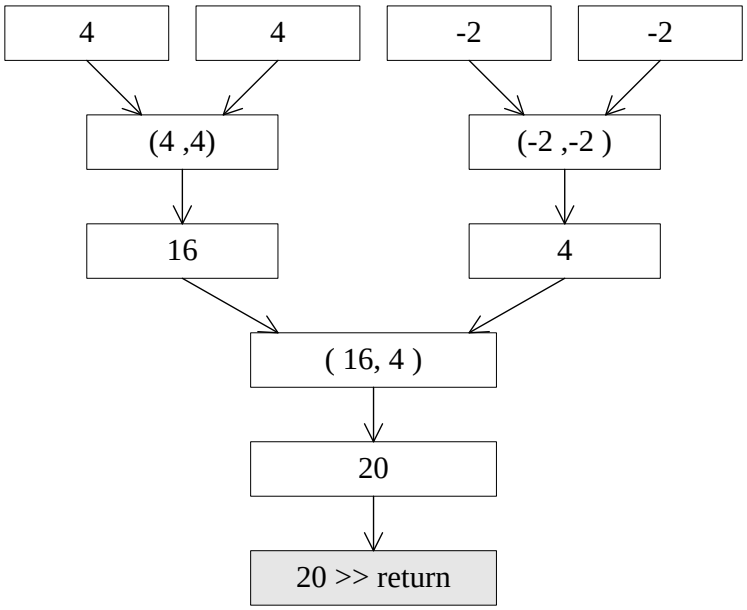


Рисунок 3.8 - Последний шаг отладки

В ходе последующих шагов осуществляются: формирование списка над результатами операции умножения, сложение этого списка и возврат результата

выполнения функции. Слой 5, определяющий полученный результат равный 20, ведет к шагу отладки, приведенному на рисунке 3.8.

Функция, после завершения отладки, выглядит следующим образом:

summa << funcdef (4,-2) { *20* >> return ;}

3.4 Примеры отладки в режиме проверки формул

Рассмотрим процесс отладки функции из предыдущего параграфа в режиме проверки формул. Добавим требования к трем узлам графа (рисунок 3.9). Где **(NODE,0):>** означает, что значение вершины-оператора больше нуля, и **(NODE,0):=** означает, что значение вершины-оператора равно нулю.

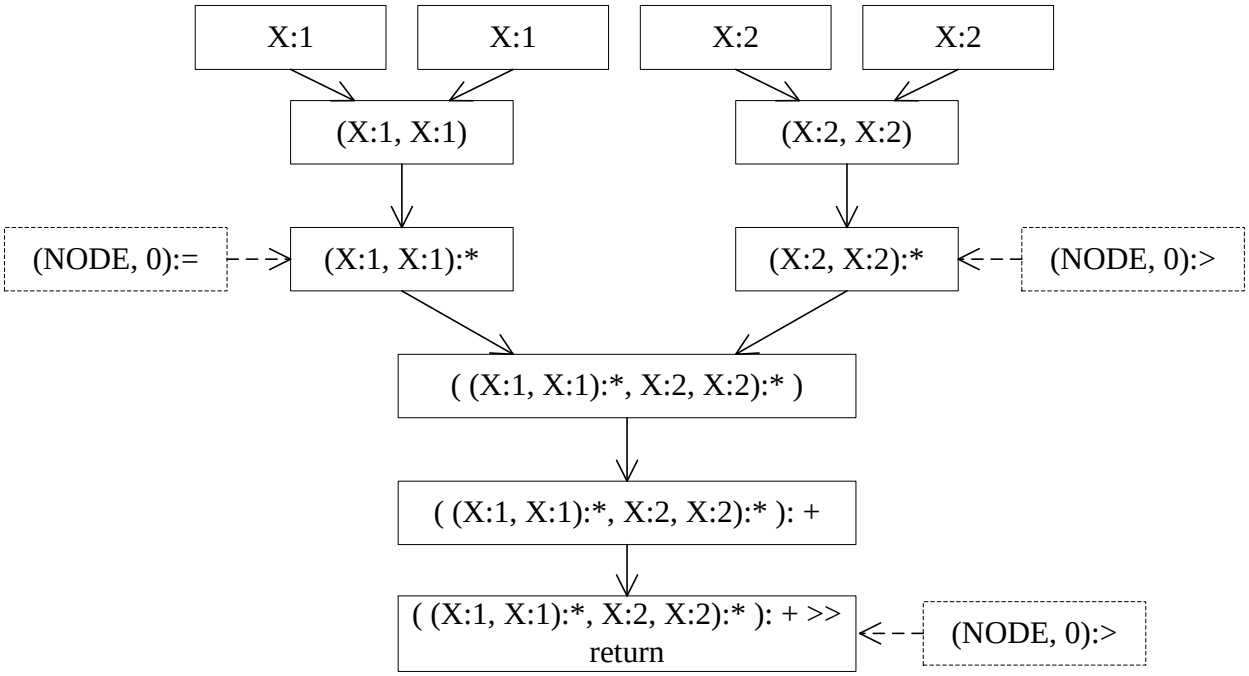


Рисунок 3.9 - Условия пользователя в режиме проверки формул

Функция не рекурсивная, поэтому отладка проходит за один шаг (рисунок 3.10), на котором вычисляются значения операторов и выражений, приписанных к узлам. В зависимости от значений выражений расцвечиваются узлы графа, с истинными или ложными утверждениями пользователя. По нажатию на вершину с приписанными условиями, можно увидеть условия и их значения, так для результирующего узла графа: **(NODE,0):> → (20,0):> → true**.

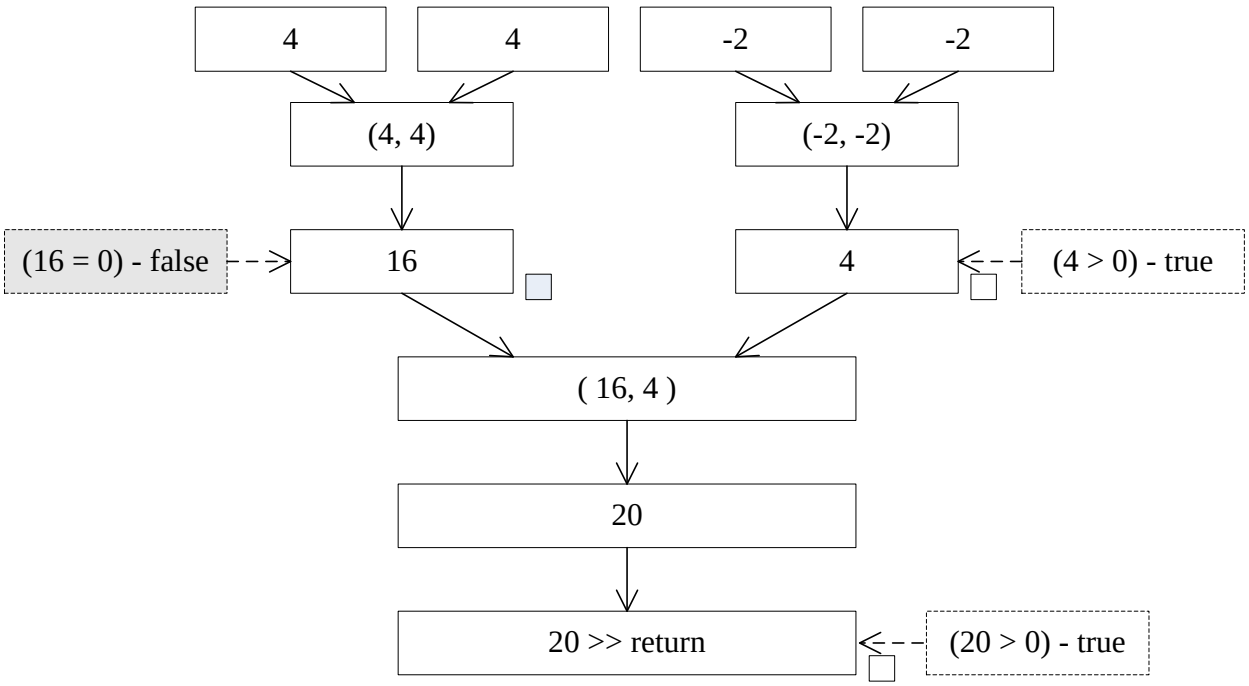


Рисунок 3.10 - Вычисление условий в режиме проверки формул

К каждому узлу графа может быть приписано несколько выражений, при этом каждое выражение формируется с помощью операторов языка программирования и может отображать различные требования, например формула **((NODE,(0, NODE:|):dup):#:[]:>):**+ значит, что результат оператора является списком, в котором существует хотя бы один элемент больше нуля.

Функция SqrtR (рисунок 3.11) рекурсивно вычисляет корень из заданного числа:

```
SqrtR << funcdef P
{A << P:1;
 X << P:2;
 Xn << (X,(((X,X):*,A):-,(2,X):*):/):-;
 ({(A,Xn)},{(A,Xn):SqrtR}):[((Xn,X):-:Abs,0.00001):(<,>=):?]>>return;
};
```

Аргумент функции P это список из двух чисел, где первое число это значение, корень из которого вычисляется (далее идентификатор A), а второе число это приближенное значение корня (далее идентификатор X). Идентификатор Xn содержит новое вычисленное значение корня, функция возвращает список (A, Xn) или осуществляет рекурсивный вызов над аргументом

(A, Xn) в зависимости от значения модуля разности Xn и X. К узлу графа, вычисляющему Xn, добавлена формула ((NODE,NODE):*, ARG:1):=. Здесь NODE – значение узла графа, оно равно Xn и ARG – аргумент функции на отдельной итерации рекурсии, то есть ARG:1 это A. Введенная формула вычисляет Xn*Xn=A. Пусть аргумент функции (16,8), шаг отладки выполняет итерацию рекурсии и вычисляет добавленную разработчиком формулу на каждом шаге (рисунки 3.12, 3.13).

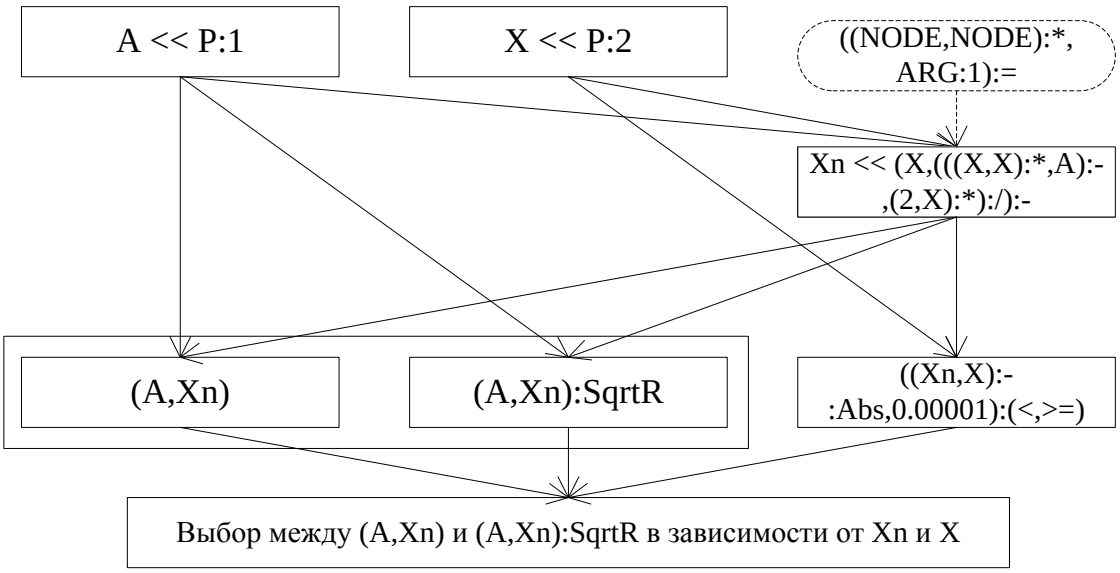


Рисунок 3.11 – Сокращенный граф функции SqrtR

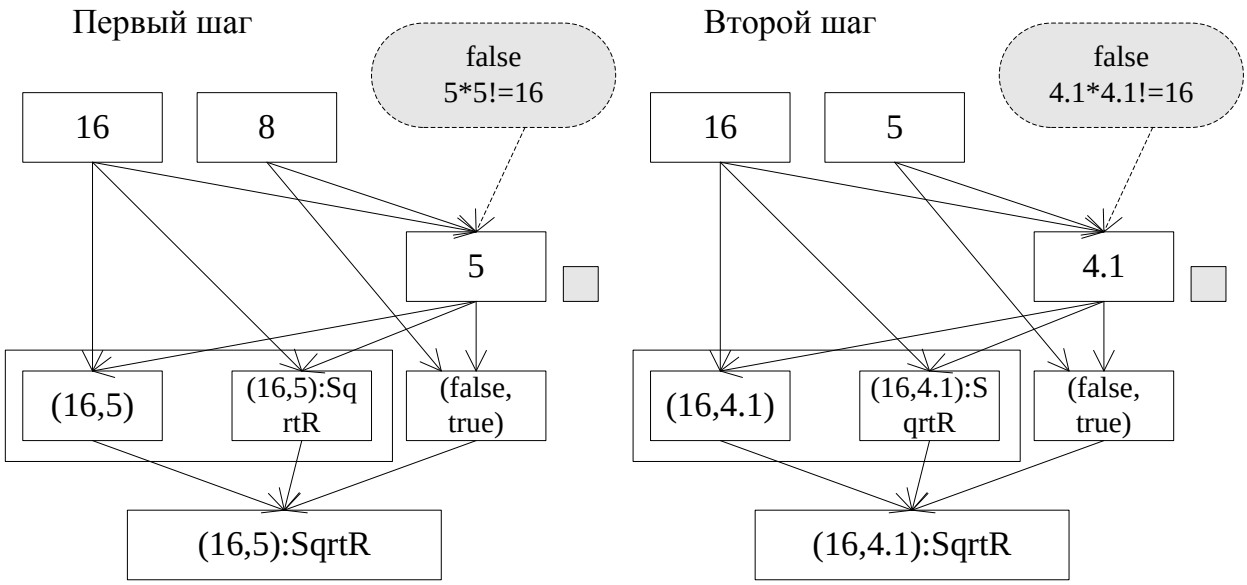


Рисунок 3.12 – Первый и второй шаги отладки функции SqrtR

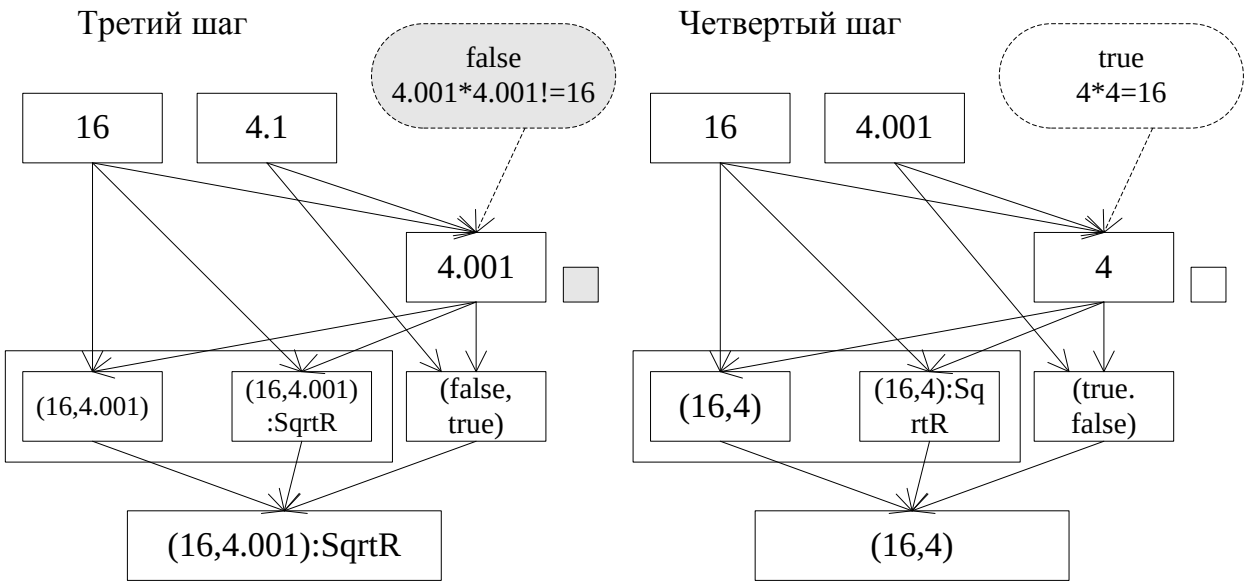


Рисунок 3.13 – Третий и четвертый шаги отладки функции SqrtR

Другие примеры отладки ФПП программ в инструментальной среде можно увидеть в Приложении А.

3.5 Возможности инструментальной среды для отладки функционально-потоковых параллельных программ

Инструментальная среда для отладки предоставляет четыре режима [34, 120, 121]: пошаговую отладку, отладку слоев, отладку ветвей и проверку формул. Совокупность режимов отладки, способов текстового представления отладочной информации и возможности визуализации графа программы определяют следующие особенности отладки.

Прохождение параллельной программы.

1 Путь выполнения параллельной программы выбирается автоматически. Эта особенность присуща всем описанным режимам отладки. Пошаговая отладка выполняет один случайный оператор из множества операторов, готовых к выполнению. Отладка слоев выполняет за один шаг все готовые к выполнению операторы, отладка ветвей выбирает случайную ветвь операторов-вершин графа из множества ветвей, способных быть вычисленными.

Режим проверки формул вычисляет все операторы функции или все операторы, составляющие итерацию рекурсивной функции на каждом шаге.

2 Путь выполнения параллельной программы выбирается запросом к пользователю и рассмотрение всех путей выполнения параллельной программы не применяются для отладки. Возможность выбора пути пользователем может быть реализована в пошаговой отладке и отладке ветвей, но различный порядок выбора операторов для выполнения не приведет к различным значениям, вычисленным операторами.

Распределение шага отладки.

1 Шаг отладки выполняется в одном параллельном участке программы. Такой режим шага поддерживается в пошаговой отладке и отладке ветвей.

2 Шаг отладки выполняется во всех параллельных участках программы. Отладка слоев и режим проверки формул обладают такой спецификой.

3 Шаг отладки выполняется в отмеченных пользователем параллельных участках программы - не применяется для отладки.

Режим проверки формул определяет шаг отладки как множество всех операторов отдельной функции, если функция рекурсивная то, как множество операторов, составляющих итерацию рекурсии.

Контрольные точки.

Точки останова и точки наблюдения не используются как механизмы контроля шага отладки. Но для отладки функционально-потоковых параллельных программ используется концепция контрольной точки с условием, то есть возможность выбрать любой оператор и приписать к нему произвольное число условий, отличие состоит в том, что при ложности или истинности условий не происходит остановка отладки, вместо этого осуществляется разметка графа программы. Это выполнено в режиме проверки формул.

Использование отладочных значений.

1 Отладочные переменные не применяются как дополнительные идентификаторы, вводимые разработчиком во время отладки, но в режиме

проверки формул можно составить и вычислить произвольное число выражений на языке программирования для каждого отдельного оператора.

2 Изменение значений переменных программы пользователем во время отладки и опции отладки частично не законченных программ не реализованы в отладчике.

Направление хода отладки.

Обратный ход не реализован как возможность перейти на предыдущий шаг отладки, но все описанные режимы отладки, используемые совместно с режимами текстового отображения «список функций» или «операторы» сохраняют информацию обо всех предыдущих шагах отладки, эта информация может быть просмотрена на любом шаге отладки. При подключении визуализации графа программы, граф также запоминает и предоставляет информацию на любом шаге обо всех вычисленных операторах, но при переходе к выполнению новой функции или следующей итерации рекурсии, доступ к предыдущим шагам отладки будет потерян. Это не относится к режимам текстового отображения.

Графическое представление процесса отладки.

1 Визуализация графа программы может быть выполнена при любом режиме распределения шага отладки, граф программы является интерактивным, вершины предоставляют информацию о записи оператора в программе и его вычисленном значении, в режиме проверки формул любая вершина графа может дополнительно содержать несколько пользовательских выражений и их вычисленные значения. Граф программы представляется как совокупность графов отдельных функций.

2 Визуализация синхронизации связана с отображаемым графом программы и связями между вершинами-операторами, также вершины, соответствующие задержанным операторам помечаются другим цветом, чем вычисленные или не вычисленные и не задержанные.

3 Визуализация ошибок реализована только для режима проверки формул, в котором вершины, содержащие ложные пользовательские выражения, дополнительно помечаются.

4 Визуализация списков и статистической информации не выполнена в среде для отладки.

Таблица 2 – Возможности отладки реализованной среды разработки и набора существующих отладчиков программ с неявным параллелизмом.

Особенности отладки	Среда	SFP	HOOD	HsDebug	Hat	Buddha
1	2	3	4	5	6	7
Прохождение параллельной программы:						
Автоматический выбор	+	+	+	+	+	+
Запрос к пользователю		+				
Все возможные пути						
Распределение шага отладки:						
Один парал. участок	+	+	+	+	+	+
Все парал. участки	+	+				
Выбранные пар. участки		+				
Все операторы функции	+					
Контрольные точки:						
Точки останова	+	+		+		
Точки наблюдения	+	+				
Использование отладочных значений:						
Отладочные переменные	+					
Изменение значений						
Отладка не законч. кода		+				
Направление хода отладки:						
Прямой ход	+	+	+	+	+	+
Прямой и обратный ход						+
Графическое представление процесса отладки:						
Граф программы	+	+				+
Списки и данные			+			
Ошибки	+				+	+

3.6 Выводы

Реализована среда для отладки функционально-потоковых параллельных программ, предоставляющая четыре режима распределения шага отладки, различные способы текстового отображения отладки, визуализацию интерактивного графа программы, вычисление пользовательских выражений во время отладки и разметку ошибочных вершин-операторов графа по указанным требованиям (условиям).

Режимы распределения шага отладки позволяют, как сосредоточится на вычислении каждого оператора программы, так и вычислять наборы готовых к выполнению операторов, показывая уровень параллелизма программы и сокращая число шагов, требуемых для отладки. Возможность добавления и проверки формул для любых операторов позволяет задать и проверить на конкретных начальных данных требования к вычислениям программы. Возможности визуализации наглядно представляют выполнение программы, уровень ее параллелизма, синхронизацию между операторами, соответствие вычислений программы требованиям пользователя. Предложенные опции повышают эффективность процесса отладки функционально-потоковых параллельных программ.

4 Верификация функционально-потоковых параллельных программ

Для верификации ФПП программ предлагаются приемы, основанные на методе индуктивных утверждений и методе индукции, использующие информационные графы исследуемых функций [17, 119, 120]. Рассматриваются аспекты, влияющие на завершимость ФПП программ. Предлагается форма задания спецификации функционально-потоковых параллельных программ [119], применяемая в методах доказательства теорем. Описываются возможности верификации ФПП программ без задания спецификации и вычисления операторов программы, а также применение метода проверки модели к ФПП программам с асинхронными списками [120].

4.1 Верификация функционально-потоковых параллельных программ без их выполнения и спецификации

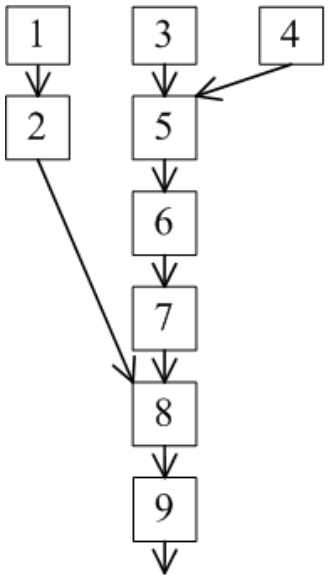
Имея текст ФПП программы, можно сделать некоторые заключения об ее свойствах без непосредственного выполнения программы и требования спецификации от пользователя.

Для этой цели можно использовать граф ФПП программы (рисунок 4.1), в котором вершины являются операторами, а дуги определяют связи между ними, и его матрицу или список смежности.

В матрице смежности информационного графа i-я строка отображает, какие операторы требуют получения данных от i-го оператора, и i-й столбец показывает, данные от каких операторов требуются для выполнения i-го оператора. Оператор, столбец которого в матрице состоит из одних нулей, является оператором, все входные данные которого известны и не нуждаются в вычислении.

Если оператор в матрице смежности представлен строкой, состоящей из одних нулей, и оператор не является оператором возврата значения из функции или оператором вывода значения, то такой оператор вычисляет нигде не

используемое в дальнейшем значение и возможно является бесполезным. Автоматически и без вычисления самой программы можно установить и предоставить пользователю все такие операторы и более того все ветки операторов, ведущих к ним, перебрав пути из выделенного оператора в обратном направлении, то есть по столбцам. Таким образом, пользователю будет представлена информация обо всех частях программы, вычисляющих неиспользуемые и неотображаемые значения.



	1	2	3	4	5	6	7	8	9
1	0	**1**	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	**1**	0
3	0	0	0	0	**1**	0	0	0	0
4	0	0	0	0	**1**	0	0	0	0
5	0	0	0	0	0	**1**	0	0	0
6	0	0	0	0	0	0	**1**	0	0
7	0	0	0	0	0	0	0	**1**	0
8	0	0	0	0	0	0	0	0	**1**
9	0	0	0	0	0	0	0	0	0

Рисунок 4.1 – Граф и матрица смежности функции Abs из раздела 2.1.1

Оператор может быть задержанным, тогда он выполняется только по требованию результата его вычисления другим оператором. В методе, предложенном выше, ответ на вопрос, является ли оператор в матрице задержанным или нет, не важен. Задержанный или нет, оператор все равно служит для вычисления неиспользуемого значения, отличие состоит в том, что «бесполезный» задержанный оператор будет вычисляться не при всех входных данных программы.

Однако появляется возможность проверять задержанные ветки операторов автоматически без непосредственного выполнения программы, используя промежуточное представление с матрицей смежности, с целью ответа на вопрос: существует ли оператор раскрытия задержанного списка? Если такой оператор отсутствует, то операторы в задержанном списке не будут вычисляться ни при каких начальных данных программы. Следовательно, они не занимают ресурсы

машины как «бесполезные» операторы, описанные выше, но они занимают часть текста программы, никогда не выполняясь.

Выявление таких задержанных операторов состоит в определении с помощью матрицы смежности, существует ли путь от задержанного оператора или ветви задержанных операторов до оператора интерпретации. Наличие такого пути не гарантирует, что существуют такие начальные данные программы, при которых выполнятся задержанные операторы. Но отсутствие пути, говорит о невозможности открытия задержанных операторов при любом выполнении программы.

Если рассматривать не отдельную функцию, а всю программу, то в этом случае задержанный список может возвращаться функцией в качестве параметра и раскрываться в вызывающей функции. Таким образом, можно либо анализировать работу с задержанными списками на общей матрице смежности графов всех функций, либо ограничиться разбором матриц смежности функций по отдельности и предупреждать о наличии или отсутствии полной раскрываемости задержанных списков как частном свойстве отдельных функций.

Исследование корректности работы с задержанными списками можно улучшать, дополнительно отслеживая уровень вложенности задержанных списков и число операций интерпретации над ними.

4.2 Проверка завершимости функционально-потоковой параллельной программы

Завершение последовательной или параллельной программы может не состояться по следующим причинам:

- обращение к функции или оператору со значением аргумента вне области определения;
- тупики, возникающие при работе с механизмами пересылки сообщений или механизмами синхронизации;

- выполнение бесконечной последовательности операторов цикла или рекурсии.

Обращение к функции или оператору со значением аргумента вне области определения – возможная ситуация в ФПП языке. Но она не приводит к зависанию программы или аварийному выходу из нее. Это связано с тем, что в ФПП модели вычислений ошибка является предопределенным значением, таким же, как число, строка или список. Любой оператор, получающий на выходе ошибку, отсылает ее на вход связанного с ним оператора. Оператор, получивший в качестве входного данного ошибку, обрабатывает ее и выдает результат, в большинстве случаев также содержащий ошибку (рисунок 4.2).

На рисунке 4.2 представлена функция, вычисляющая выражение 5x + 5/x. В качестве аргумента функции x выберем ноль (рисунок 4.2).

```
func << funcdef x
{ ((x, 5):*, (5, x):/):+ >> return ; }
```

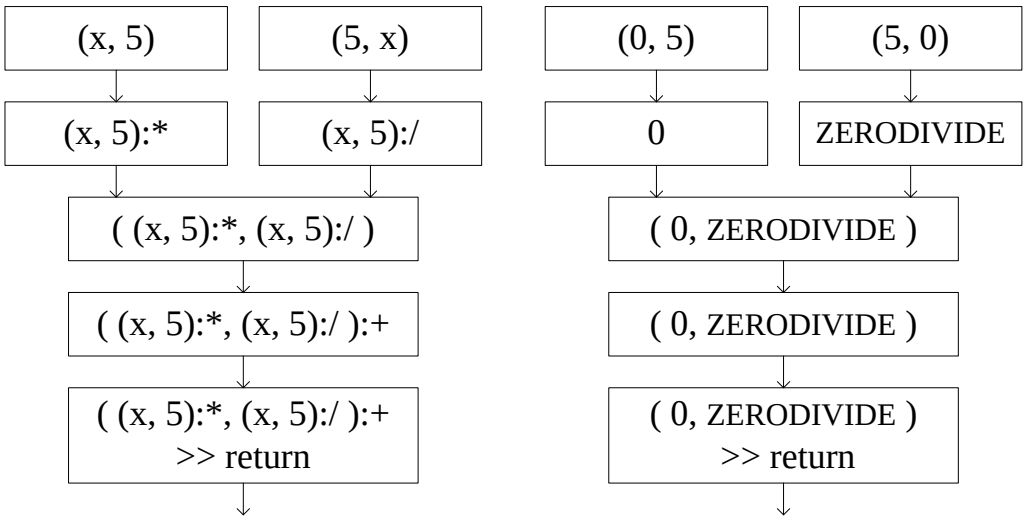


Рисунок 4.2 - Граф функции с программными операторами и вычисленными значениями

Тупики, возникающие при работе с механизмами пересылки сообщений или механизмами синхронизации, в ФПП программах невозможны по причине отсутствия таких механизмов в ФПП модели. Функционально-потоковый язык оперирует дополнительным механизмом – задержанными списками. Если над задержанным списком не выполняется операция открытия, он передается на вход

других операторов как неопределенное задержанное выражение – expression, которое либо будет открыто позже, либо даст в качестве результата работы части операторов ошибку. Таким образом, наличие задержанного списка не может привести к бесконечному ожиданию завершения программы.

Возможна более сложная ситуация, когда задержанный список включает в себя вычисление идентификаторов, и эти идентификаторы используются в программе далее. Здесь результат выполнения программы зависит от правил обработки описанной ситуации. Возможны три варианта.

1 Все идентификаторы в задержанном списке считаются локальными, их область действия ограничивается тем задержанным списком, в котором они объявлены. Результатом выполнения такой программы будет ошибка на этапе трансляции вида «неизвестный идентификатор».

2 Идентификатор, заключенный в задержанный список, вычисляется, если задержанный список открывается, либо если существует запрос на этот идентификатор от вычисляемого оператора. При выполнении такой программы произойдет вычисление задержанного списка и его идентификаторов.

3 Идентификатор, заключенный в задержанный список, вычисляется только при открытии задержанного списка. Здесь возникнет ситуация бесконечного ожидания значения идентификатора, если задержанный список не открылся.

Последний вариант обработки идентификаторов в наборе задержанных операторов способен привести к бесконечному ожиданию, и менее предпочтителен, чем два предыдущих. Проверка наличия запроса задержанных идентификаторов от операторов, не включенных в задержанный список, может быть выполнена автоматически без непосредственного выполнения программы, так как это предлагается в разделе 4.1 с помощью списка операторов и матрицы смежности. Найденные запросы могут быть представлены пользователю, также как «бесполезные» операторы и никогда невычисляемые задержанные операторы.

Выполнение бесконечной последовательности операторов цикла или рекурсии – распространенная причина незавершения программы.

Функционально-потоковый параллельный язык не содержит циклов, но использует рекурсии, которые влияют на свойство завершимости программы.

Проверка завершимости рекурсивной ФПП программы без ее выполнения состоит в выявлении следующих ситуаций:

- заключен ли рекурсивный вызов в задержанный список, если нет, то рекурсия будет бесконечной при любых входных данных, так как оператор, не входящий в задержанный список, будет выполняться в любом случае, и это распространится на всю последовательность рекурсивных вызовов;
- проверка наличия аргумента рекурсивной функции в списке аргументов рекурсивного вызова, если аргумент функции присутствует в списке, то при любых начальных данных рекурсия будет бесконечной.

Проверка завершимости рекурсивной ФПП программы совместно с ее выполнением над указанными начальными данными может проходить автоматически. Она состоит в запоминании всех аргументов рекурсивных вызовов, и ситуация повторения аргумента для одной и той же рекурсивной функции обязательно приведет к бесконечной последовательности рекурсивных вызовов. Недостаток такого подхода состоит в том, что результаты проверки действительны только для конкретных указанных начальных данных программы.

Среди методов доказательства теорем, осуществляющих проверку для всех возможных начальных данных, для аналитической верификации можно выделить функцию декремента [76]. Автоматизация такого подхода затруднительна, так как требует обширную базу правил, автоматический преобразователь формул и взаимодействия с квалифицированным пользователем.

4.3 Верификация функционально-потоковых параллельных программ с асинхронными списками

Модель ФПП вычислений [66, 71, 74, 85] исключает зависимость результатов от порядка выполнения операторов, готовых к вычислению. Исключение составляет ситуация, возникающая при использовании асинхронных

списков [62, 64, 72]. Обычный список ожидает формирования всех его элементов, тогда как асинхронный список выдает первый сформированный элемент (операция «***1***») и список, состоящий из всех оставшихся элементов (операция «***-1***»), часть которых может быть еще не вычислена. Таким образом, если время вычисления элементов асинхронного списка различается мало, то при разных запусках программы порядок получения данных из списка может измениться, в связи с чем, возникает вопрос о поведении программы при различном порядке обработки элементов из асинхронного списка.

Верификация ФПП программы состоит в переборе всех возможных комбинаций обработки асинхронного списка для каждой операции выбора элемента. При этом программа выполняется над указанными начальными данными. Проверка осуществляется подстановкой конкретных данных в операторы программы. Спецификация от пользователя не требуется. При рассмотрении асинхронного списка в режиме верификации сначала вычисляются все его элементы, потом производятся вычисления всех возможных комбинаций изъятия. Во время отладки асинхронный список рассматривается как обычный список. Чем больше размерность асинхронного списка и чем чаще встречается операция изъятия из него, тем большее число вариантов исполнения программы существует.

Режим верификации позволяет рассмотреть все возможные варианты обработки асинхронного списка и выделить пути, интересные разработчику. Цель верификации – определить, приводят ли все возможные пути выполнения асинхронной программы к одинаковому результату, или существуют пути, вычисляющие разные значения. При этом пути, приводящие к разным результатам, могут быть детально представлены пользователю.

Верификация ФПП программ, базирующаяся на переборе комбинаций данных, поступающих в асинхронные списки, позволяет проверить зависимость вычислений от последовательности данных и тем самым выявить ряд программных ошибок, связанных с этой особенностью языка программирования.

4.3.1 Пример верификации функционально-потоковой параллельной программы с асинхронным списком

Рекурсивная функция получает асинхронный список и операцию плюс или минус. В первом случае функция вычисляет выражение A1+A2-A3+A4-A5+…, во втором A1-A2+A3-A4+A5-… Команды, связанные с асинхронным срабатыванием, возможно вычисляющие различные значения при различных запусках программы, отмечены в тексте ниже подчеркиванием.

```
F << funcdef A {
// Формат аргумента: A=(asynch(x1, x2, ... , xn),op)
// где x1, x2, ... , xn - числа, n>=2,  а op - плюс или минус
x << A:1;     // выделение асинхронного списка в x
op << A:2;   // выделение операции в op
x1<<x:1;
//выделение первого готового числа из асинхронного списка x в x1
tail_1<< x:-1;
// выделение оставшихся значений асинхронного списка в tail_1
// проверка - есть ли элементы в в tail_1
 [((tail_1:|, 0):[=, !=]):?]^
        (  x1:op,  //если элементов в tail_1 нет, то x1 возвращается функцией
           {   // если элементы в tail_1 есть
             block {
               x2 << tail_1:1; // выделение первого готового числа из tail_1  в x1
               s << (x1,x2):op;      // операция над двумя готовыми элементами
               tail_2<< tail_1:-1;  // оставшиеся элементы асинхронного списка
              // изменим плюс на минус и наоборот
                [((op,+):[=, !=]):?]^( { op << - ; }, { op << + ; } ) ;
 // рекурсивная обработка оставшихся элементов асинхронного списка
               [((tail_2:|, 0):[=, !=]):?]^
               ( s, { (asynch( s, tail_2:[]),op):F } ):. >>break   }
```

}):. >>return; }

В качестве начальных данных применяется список вида **(asynch(1,2,3), -)**. Рассмотрим все возможные пути выполнения программы (рисунок 4.3).

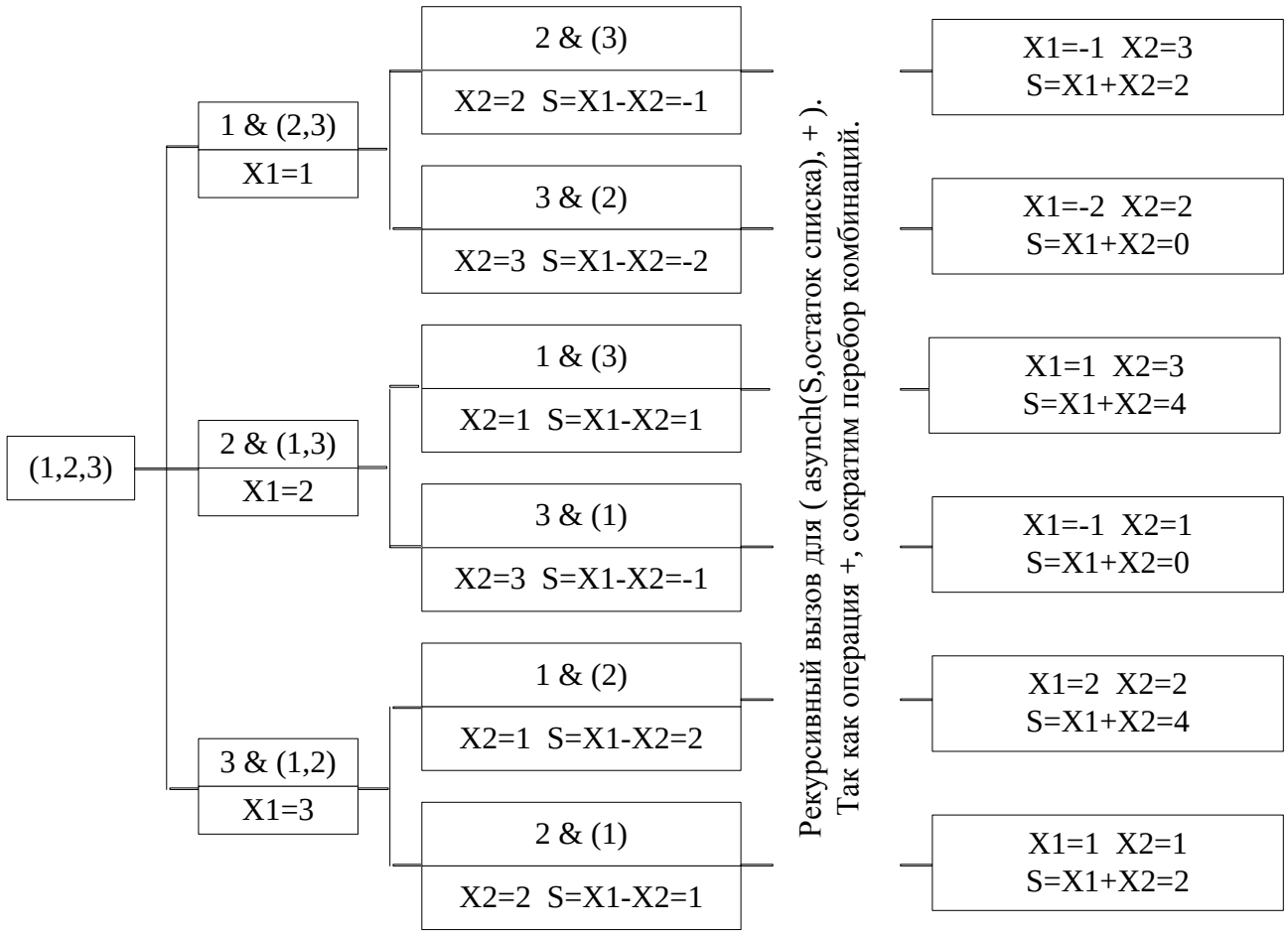


Рисунок 4.3 - Сокращенные пути выполнения программы

Режим верификации позволит установить, что различная последовательность готовности элементов в списке приведет к разным вычисленным результатам функции S (0 или 2 или 4), а также отобразит все возможные варианты обработки асинхронного списка в текстовом виде.

Если приведенную функцию заменить подобной функцией, вычисляющей A1+A2+A3+A4+A5+…, перебор значений асинхронного списка покажет, что результат вычислений всегда одинаков, асинхронность не нарушает логику вычислений и не влияет на результат.

4.4 Инструментальная поддержка методов верификации со спецификацией пользователя

Для верификации ФПП программы применяется спецификация, описывающая общий вид входных данных программы посредством стандартных типов данных, таких как списки, строки и числа, и специальных утверждений и констант. Множество утверждений спецификации выглядит следующим образом.

~unknownnumber – неизвестное число.

~unknownbool – неизвестное логическое значение (ложь или истина).

~unknown – неизвестное значение, может быть числом, текстом, логическим значением, списком, строкой или любым другим данным программы. Использование константы для спецификации имеет малую пользу, константа ~unknown часто является результатом выполнения программного оператора над данными, не определенными во множестве правил для верификации программы.

~gt A – число большее чем указанное число A.

~lt A – число меньшее чем указанное число A.

~ge A – число больше либо равное указанному числу A.

~le A – число меньше либо равное указанному числу A.

~A interval B – число, лежащее в указанном интервале [A, B].

Таким образом, спецификация начальных данных может иметь, например следующий вид: (5, ~lt 0, ~unknownbool) – список, первый элемент которого равен пяти, второй меньше нуля и третий является ложью или истиной, либо (~unknownnumber, 0) – список, первый элемент которого неизвестное число, а второй ноль.

Спецификация обязательно задается пользователем только для входных данных программы. Возможность задания спецификации для цели вычислений или для промежуточных операторов программы поддерживается только в одном из режимов верификации.

При верификации не используются правила логического вывода или преобразование формул. Вместо этого применяется набор правил, созданный для

каждого отдельного оператора функционально-потокового параллельного языка, позволяющий для известных входных данных оператора, среди которых есть хотя бы одно, заданное утверждением спецификации, поставить в соответствие результат работы оператора. Выходное значение оператора может быть известным данным, конкретным числом, строкой, списком или также являться утверждением спецификации, может являться комбинацией известных значений и утверждений спецификации. Правила обработки вычислений над спецификациями опираются на правила языка «Пифагор» и формулы интервального анализа [36, Приложение Б]. Набор правил реализован для всех функций ФПП языка «Пифагор», что является возможным благодаря их небольшому числу, но он не охватывает все возможные варианты обработки аргументов функциями. Если во время верификации при рассмотрении оператора отсутствует правило для вывода его результата работы, результатом по умолчанию является константа спецификации ~unknown. При возникновении такой ситуации в ходе верификации, управление передается человеку. Пользователю предлагается самостоятельно ввести результат выполнения оператора над входными данными. Отказ от ввода означает, что результатом становится ~unknown, если же пользователь ввел результат, он принимается истинным и используется в дальнейших вычислениях.

Режим верификации в большей мере относится к методам доказательства теорем, чем проверки моделей. Но от методов доказательства теорем его отличает отсутствие логического анализа множества выведенных утверждений, их проверки, упрощения и построения новых утверждений на основе набора существующих с помощью правил логического вывода, ссылающихся не только на правила обработки программных операторов. Такое упрощение методов доказательства теорем позволяет облегчить автоматизацию процесса верификации.

В разработанной среде для создания, отладки и верификации программ, интерфейс верификации почти совпадает с интерфейсом отладки. Для запуска верификации программы используется тот же диалог (рисунок 3.4), что и для

отладки или отдельный пункт меню «Проверка формул». И в том и в другом случае, если в поле аргумента будут записаны только известные точные данные (конкретные числа, строки) то будет произведена отладка. Если среди входных данных присутствует утверждение спецификации, то запустится процесс верификации, обрабатывающий конкретные значения с помощью запроса к интерпретатору, а утверждения спецификации с помощью поиска в наборе правил и отправляющий запрос к пользователю в случае отсутствия результата поиска.

Результаты верификации предоставляются пользователю в том же виде, что и результаты отладки, верификация происходит, как и отладка, пошагово. Способ представления результатов зависит от выбора пользователя в диалоге (рисунок 3.4), так шагом верификации может служить отдельный оператор, слой (рисунок 3.1) или ветвь (рисунок 3.2) операторов, подача результатов выполнения операторов может быть реализована как список операторов, список функций или свертка функций, также возможно отображение результатов на графе.

Если для верификации был выбран режим «Проверка формул», то появляется возможность, кроме спецификации входных данных функции добавить произвольное число условий к произвольным операторам функции (рисунок 3.9). Условия записываются как выражения на самом языке программирования (без возможности вызова функций, описанных программистом), в них дополнительно можно использовать утверждения спецификации и специальные константы, доступные только в режиме «Проверка формул». Это константы: ARG – аргумент функции, NODE – значение той вершины-оператора графа функции, к которой добавлено пользовательское условие, NODE <натуральное число> - значение оператора с указанным номером, номера назначаются операторам автоматически перед началом отладки/верификации и отображаются в интерфейсе среды. Таким образом, пользовательское условие может быть к примеру таким: **((NODE,ARG):<, (NODE,~ 0 interval 1):!=):***, то есть значение текущего оператора меньше чем аргумент функции и не совпадает с интервалом (0,1). При определении результата пользовательских выражений (условий) используется аналогичный подход, что и

для вычисления самих операторов, то есть если утверждения спецификации отсутствуют в выражении, результат вычисляется интерпретатором, иначе происходит поиск в наборе правил, если поиск не дает результата происходит запрос к пользователю. Поэтому возможна и такая ситуация, когда результат введенного пользователем условия определяется им самим.

Представленный режим верификации позволяет пользователю проследить выполнение программы над обобщенными входными данными, заданными посредством утверждений спецификации, получить результаты работы каждого программного оператора, если они есть в наборе правил, а также результат, вычисляемый программой. Анализ процесса и результата выполнения программы над обобщенными начальными данными производится самим пользователем или отслеживается с помощью дополнительных пользовательских условий, приписанных к произвольным операторам программы.

4.4.1 Пример верификации функционально-потоковой параллельной программы со спецификацией пользователя

Функция Abs получает число P и вычисляет его модуль.

Abs << funcdef P

{ ({P:-}, P): [(P,0):(<,>=) :?]:. >> return }

Пусть спецификацией начальных данных функции является ~lt 0, то есть P число меньшее нуля. Результат верификации функции для указанной спецификации будет следующим (рисунок 4.4).

Оператор {P:-} при подстановке P станет {~lt0:-}. Поиск соответствия в наборе правил даст результат {~gt0}, то есть задержанный список с положительным числом.

Оператор ({P:-}, P) при подстановке P и вычисленного для предыдущего оператора значения даст ({~gt0}, ~lt0).

Оператор (P,0) станет (~lt0, 0). Команда (P,0):(<,>=) заменится на (~lt0, 0) :(<,>=), это означает проведение двух сравнений ~lt0<0 и ~lt0>=0, поиск в наборе правил выдаст результат true и false, что сформирует список (true, false).

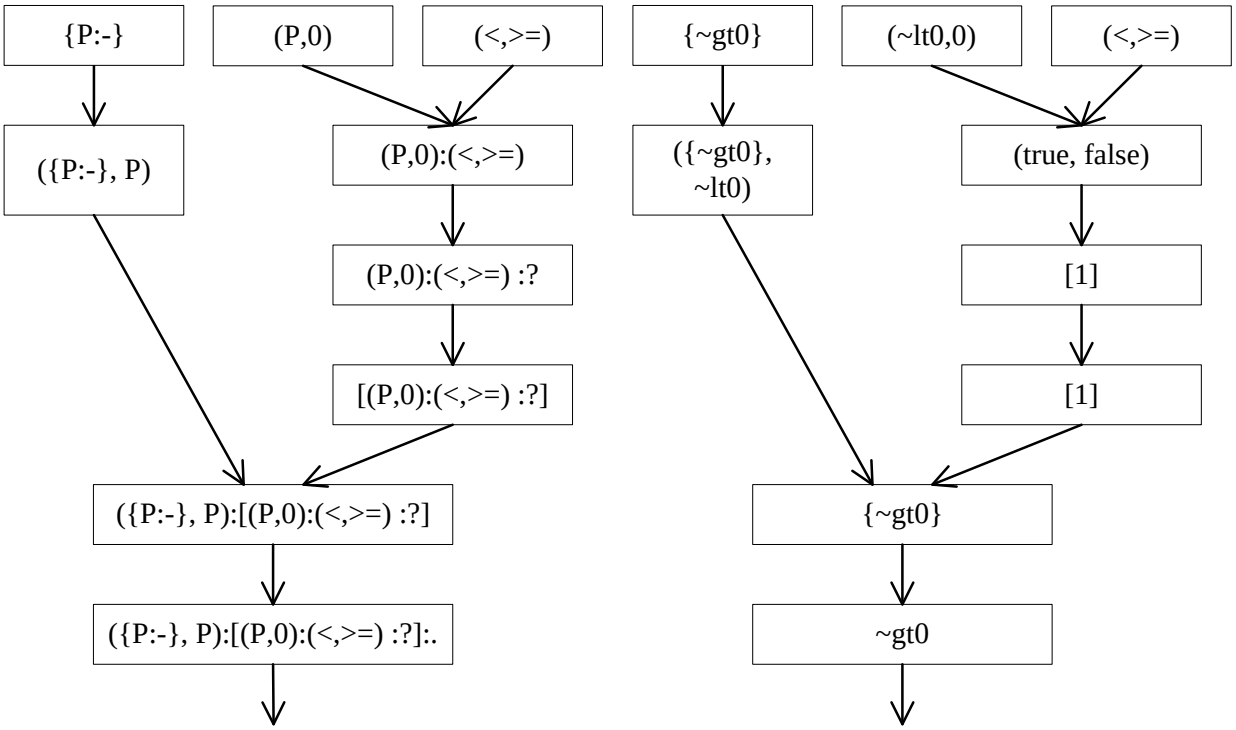


Рисунок 4.4 – Графы функции Abs с операторами и вычисленными значениями для аргумента ~lt0

Оператор (P,0):(<,>=):? после подстановки предыдущих вычисленных значений станет равен (true, false):?. Оператор (true, false):? не содержит данных, содержащих утверждения спецификации, поэтому его значение не ищется в наборе правил, а вычисляется интерпретатором. Оно равно [1], это параллельный список из индексов истинных значений в списке.

Оператор ({P:-}, P): [(P,0):(<,>=) :?] после подстановки вычисленных значений станет ({~gt0}, ~lt0) : [1], это выбор первого элемента из списка, что даст {~gt0}.

Оператор ({P:-}, P): [(P,0):(<,>=) :?]:. при подстановке станет равен {~gt0}:., что даст результат ~gt0. Последний оператор ({P:-}, P): [(P,0):(<,>=) :?]:. >> return завершит функцию с вычисленным значением ~gt0, то есть с числом больше нуля.

Таким образом, верификация над указанными спецификацией начальными данными показала, что функция Abs, получающая на входе отрицательное число, вычисляет число положительное.

Сколько операторов функции обрабатывается за шаг верификации, зависит от выбора пользователя и не влияет на вычисленные или найденные в наборе правил результаты выполнения операторов.

Если начать верификацию функции Abs, указав спецификацию входного аргумента как ~ge 0, число большее либо равное нулю, с помощью аналогичных действий будет показано (рисунок 4.5), что результат выполнения функции станет ~ge 0. То есть верификация подтвердит, что функция вычислит число положительное и при положительном аргументе.

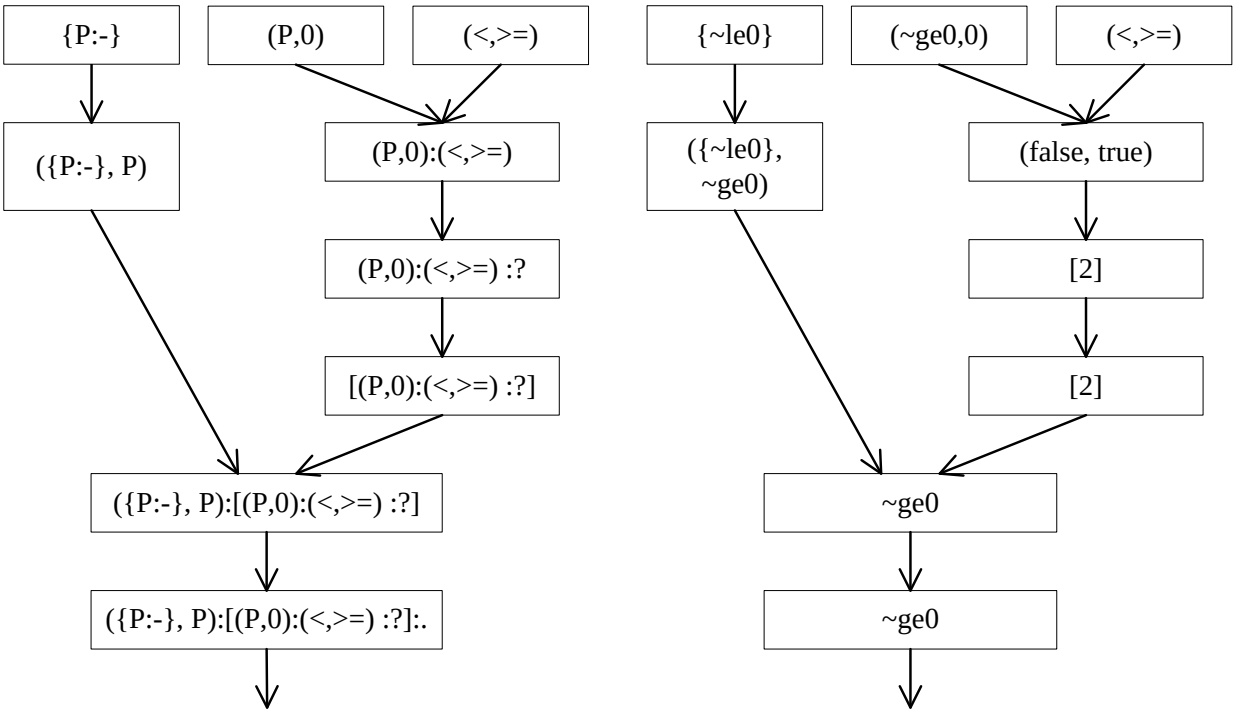


Рисунок 4.5 – Графы функции Abs с операторами и вычисленными значениями для аргумента ~ge0

Результат верификации и процесс ее прохождения зависит от спецификации входных значений, так если указать аргументом функции ~le 0, число меньше либо равное нулю, то верификация будет происходить следующим образом (рисунок 4.6).

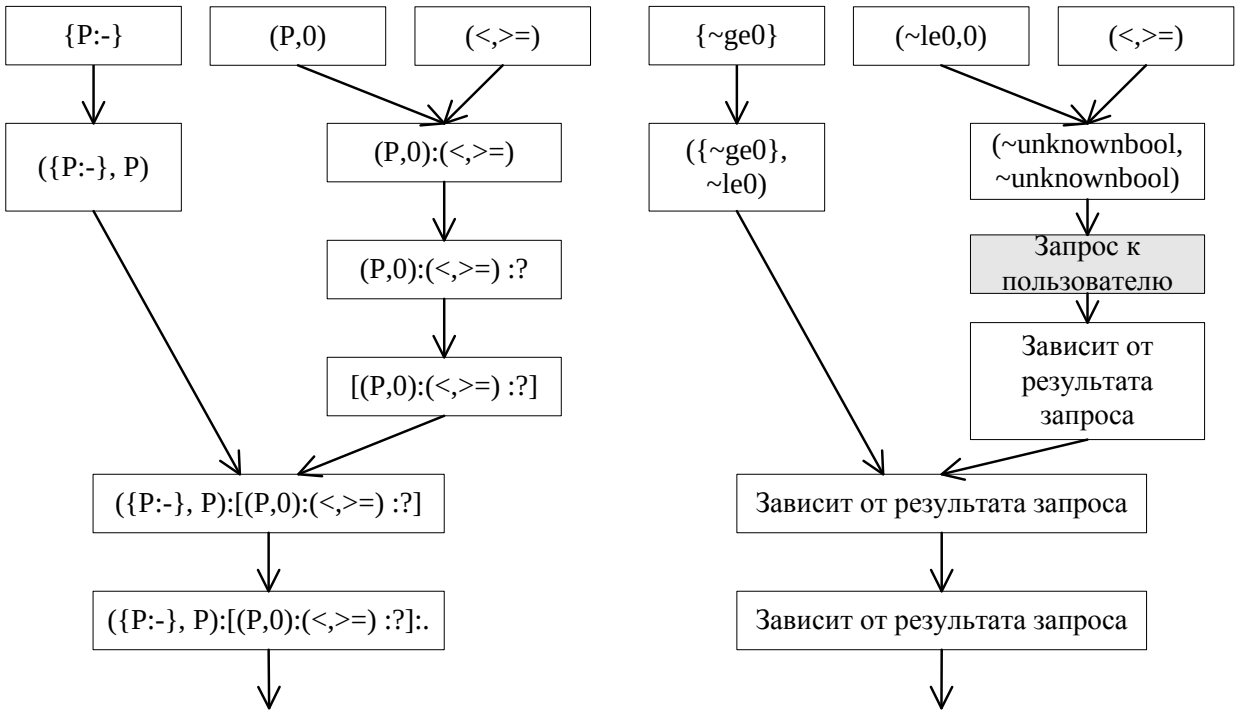


Рисунок 4.6 – Графы функции Abs с операторами и вычисленными значениями для аргумента ~le0

Оператор {P:-} при подстановке P станет {~le0:-}. Поиск соответствия в наборе правил даст результат {~ge0}.

Оператор ({P:-}, P) при подстановке P и вычисленного для предыдущего оператора значения даст ({~ge0}, ~le0).

Оператор (P,0) станет (~le0, 0). Команда (P,0):(<,>=) заменится на (~le0, 0) :(<,>=), что сформирует список (~unknownbool, ~unknownbool), так как оба сравнения могут оказаться как истинными, так и ложными.

Оператор (P,0):(<,>=):? после подстановки предыдущих вычисленных значений будет равен (~unknownbool, ~unknownbool):?. Поиск в наборе правил окажется неудачным, результат по умолчанию будет принят за ~unknown и произойдет запрос к пользователю. Если пользователь не введет результат работы оператора, то останется константа спецификации ~unknown. Если пользователь введет произвольную строку текста, то она будет принята за результат работы оператора, без проверки ее соответствия на возможность возникновения указанного результата.

Результаты работы всех оставшихся операторов функции Abs в режиме верификации будут зависеть от того, какой именно результат указал пользователь и указал ли на предыдущем шаге.

Верификация функции Abs для аргумента ~lt 0, пройдет полностью автоматически и покажет, что если аргумент функции меньше нуля, то функция вычисляет число больше нуля.

Верификация функции Abs для аргумента ~ge 0, пройдет полностью автоматически и покажет, что если аргумент функции больше либо равен нулю, то функция вычисляет число больше либо равное нулю.

Верификация функции Abs для аргументов ~gt 0 и ~le 0, на этапе рассмотрения оператора :? обратиться с запросом к пользователю, ее дальнейшее прохождение будет зависеть от ответа пользователя.

4.4.2 Верификация функционально-потоковой параллельной рекурсивной программы со спецификацией пользователя

При верификации рекурсивной функции автоматически проверяется только одна ее итерация, на которой может быть получен либо окончательный результат, состоящий из точных значений или выражений спецификации, либо оператор вызова рекурсивной функции. В последнем случае будет представлен новый аргумент для следующей итерации рекурсии. Для ее исследования пользователь должен начать процесс верификации сначала, указав полученный аргумент. Так можно рассмотреть только ограниченное число итераций. Если на какой-то из них рекурсивный вызов не будет выполнен, это покажет, что рекурсия завершается для указанного множества входных значений за известное число итераций и вычисляет полученные в процессе верификации данные. Если на всех рассмотренных итерациях происходит рекурсивный вызов, то вывод об особенностях выполнения рекурсии и возможности ее завершения пользователь может попытаться составить сам, используя множество полученных аргументов, а

также промежуточных значений, полученных при рассмотрении различных итераций рекурсии.

Рекурсивная функция Add (рисунок 4.7) получает список произвольной длины v и вычисляет сумму его элементов.

```
Add << funcdef v
{ [((v:|,1):[<,=,>]):?]^  //сравнение длины списка с единицей
   ( {.},                 //пустой список
     {v:1},               //первый элемент из списка
   /*сумма первых двух элементов списка и список v после удаления двух его
   первых элементов подаются на вход рекурсивной функции Add*/
      {((v:1,v:2):+, v:-1:-1:[]):Add:.}
   ) :. >>return;     }
```

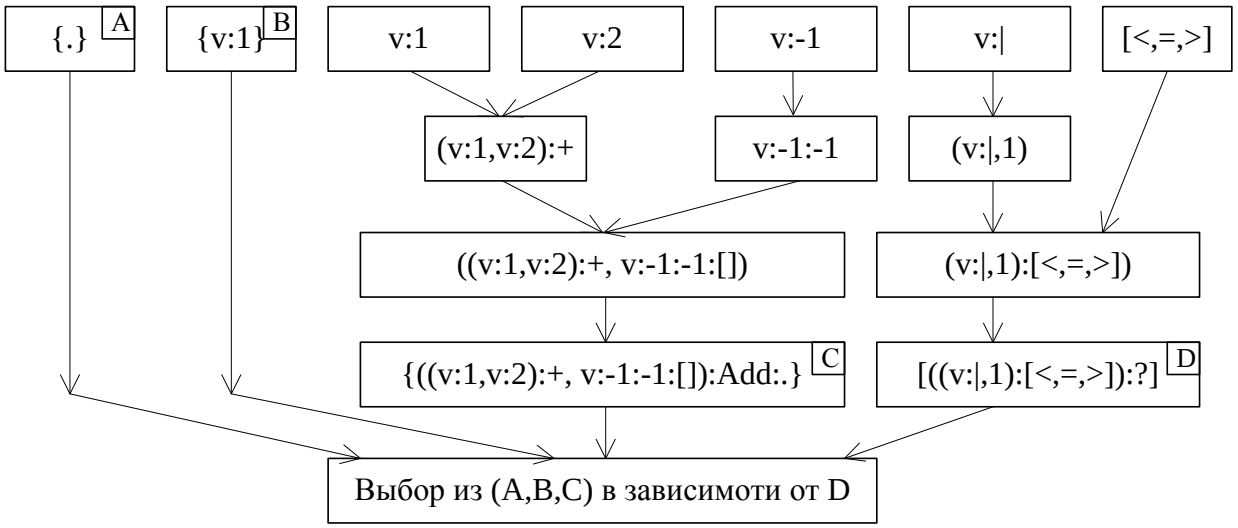


Рисунок 4.7 – Сокращенный граф функции Add

Пусть спецификацией начальных данных функции является список (~-1 interval 1, ~3 intreval 4, ~ge1), то есть первый элемент списка лежит в интервале [-1, 1], второй в интервале [3, 4] и третий больше либо равен единице.

Верификация итерации рекурсии пройдет полностью автоматически (рисунок 4.8), кроме оператора (~ 2 interval 5, ~ge1):Add. Здесь пользователю будет предоставлена запись оператора и произойдет запрос на ввод значения результата работы оператора, так как автоматический поиск в наборе правил не даст результата. Пользователь может ответить на запрос либо нет, тем не менее,

верификация автоматически покажет, что первая итерация рекурсии приводит к вызову рекурсии над списком (~ 2 interval 5, ~ge1).

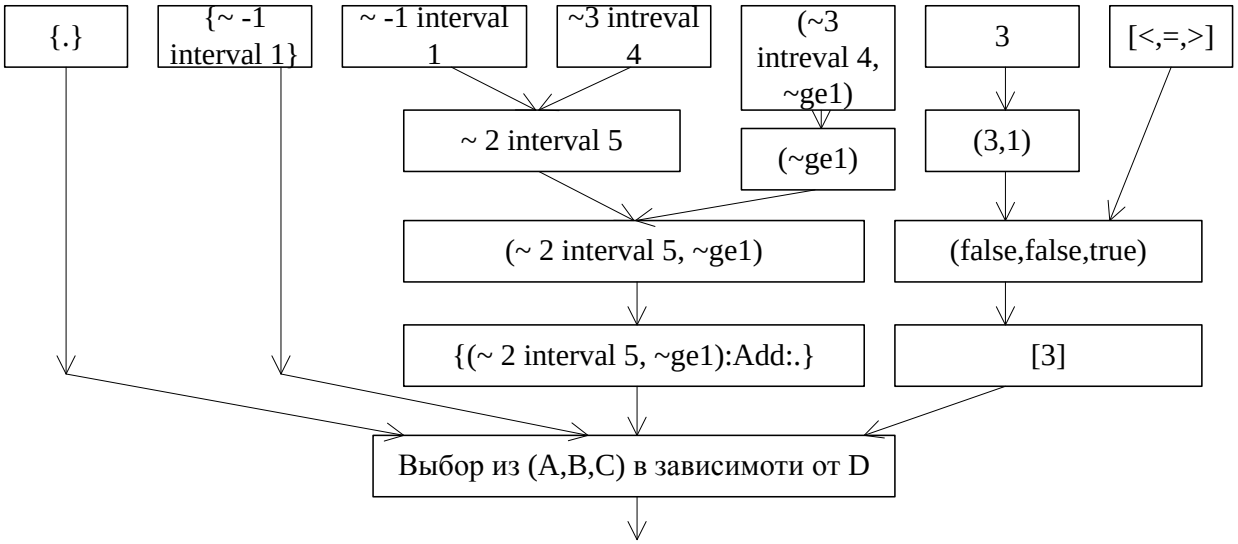


Рисунок 4.8 – Граф первой итерации функции Add

Для автоматического рассмотрения второй итерации рекурсии над списком (~ 2 interval 5, ~ge1) следует начать процесс сначала, указав спецификацию начальных данных как (~ 2 interval 5, ~ge1).

Верификация второй итерации пройдет полностью автоматически (рисунок 4.9), за исключением оператора входа в следующую итерацию рекурсии, и покажет, что третья итерация будет производиться над списком (~ge3).

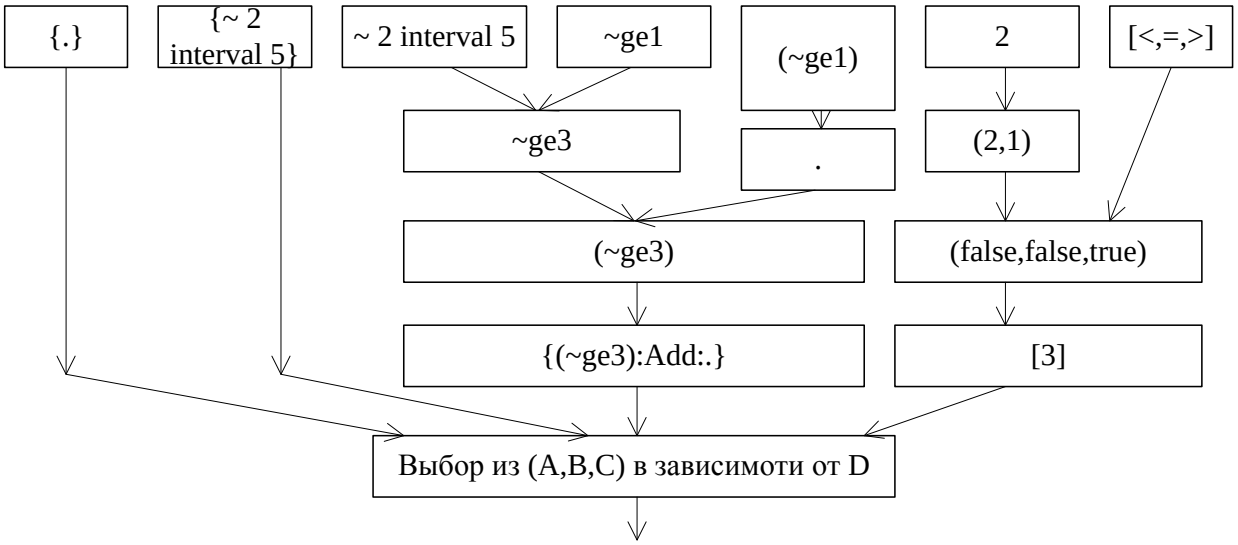


Рисунок 4.9 – Граф второй итерации функции Add

Верификация третей итерации рекурсии (рисунок 4.10) автоматически покажет, что итерация окажется последней и ее результатом станет (~ge3).

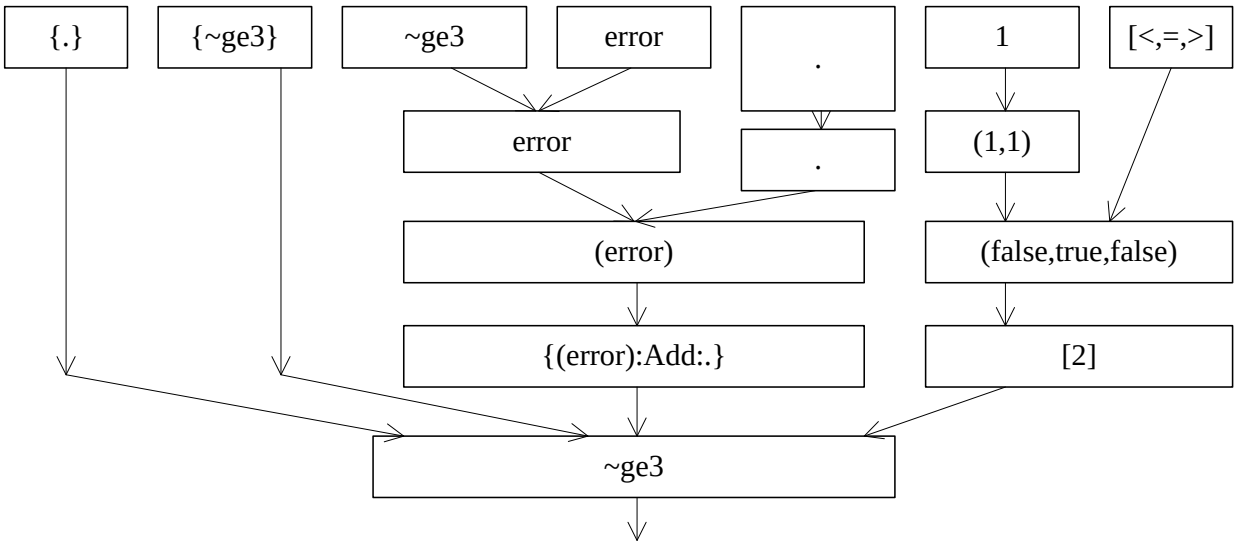


Рисунок 4.10 – Граф третей и последней итерации функции Add

Утверждения спецификации и автоматическая верификация могут быть использованы как для детального рассмотрения особенностей выполнения программы над обобщенными данными, так и для получения оценки, в каких границах лежат вычисляемые в процессе работы значения.

4.4.3 Верификация функционально-потоковой параллельной рекурсивной программы со спецификацией пользователя в режиме «Проверка формул»

Дополнительный инструмент, помогающий пользователю сделать вывод о корректности программы, – это возможность добавления к ее произвольным операторам логических формул. Здесь спецификация программы основывается не только на описании множества начальных данных, но и на утверждениях об ожидаемых свойствах вычисляемых любым оператором значений, позволяющих выделить корректные и некорректные вычисления программы. Вид добавляемых формул и расположение их на графе программы зависят от пользователя. Автоматическая верификация предоставляет вычисленные значения формул и отмечает на графе истинные и ложные формулы. На основе этих данных пользователь может либо сделать заключения о свойствах выполнения

программы, либо убедиться, в случае равенства формул булевым значениям, что программа удовлетворяет или не удовлетворяет заданным им требованиям.

Для рекурсивных функций пользовательские формулы будут заново пересчитываться на каждой итерации рекурсии. Число рассматриваемых итераций ограничено желанием пользователя. Возможна ситуация, когда требования, истинные на одних итерациях, оказались ложными или равными не булевым значениям на других итерациях. Анализ полученных в процессе верификации значений пользовательских формул для множества последовательных итераций рекурсии производится пользователем самостоятельно.

Функция Add (рисунок 4.11) должна находить сумму положительных элементов списка произвольной длины.

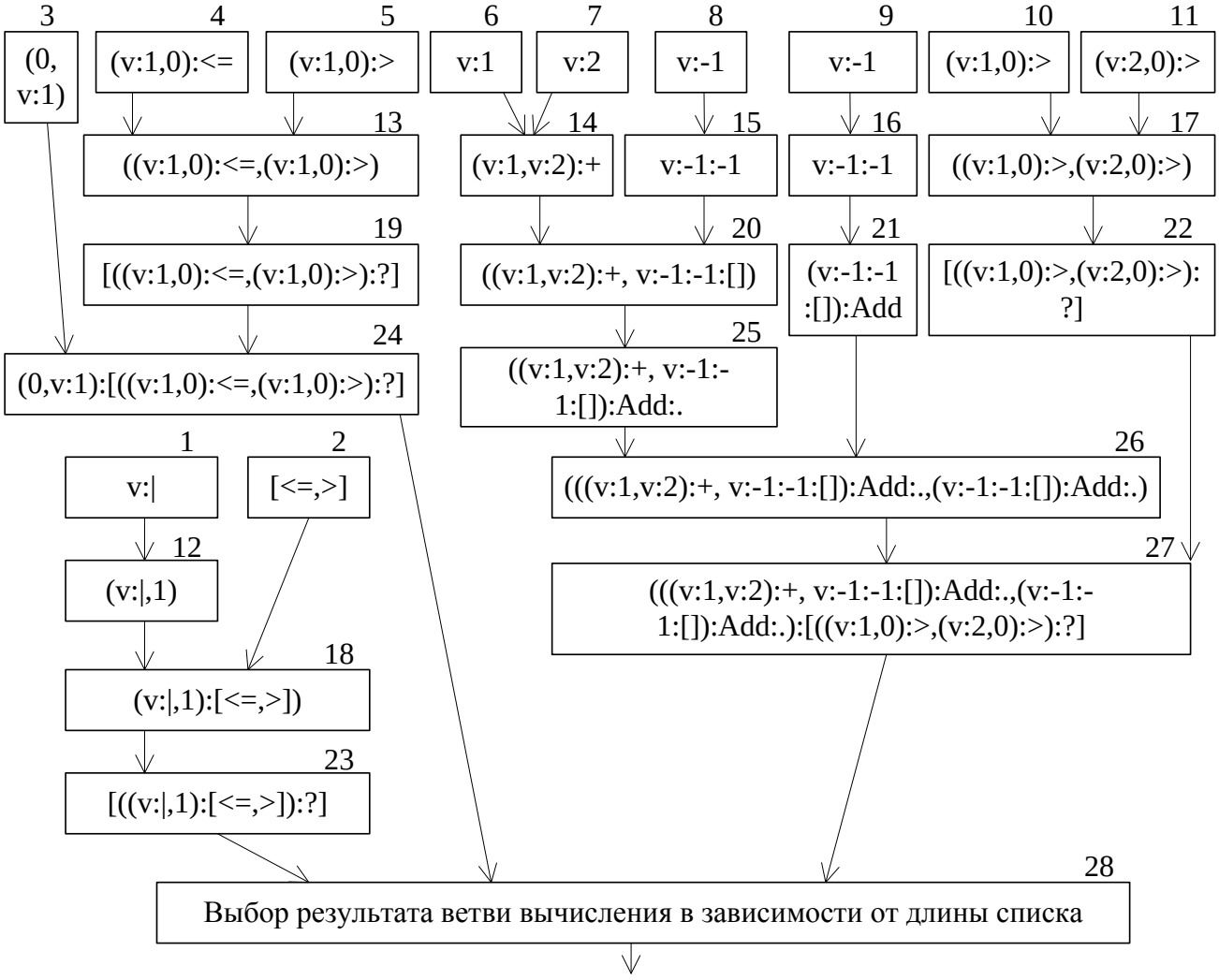


Рисунок 4.11 – Сокращенный граф функции Add

Add << funcdef v

```
{ [((v:|,1):[<=,>]):?]^      //сравнение длины списка с единицей
      //если длина списка <= 1
      ( {(0,v:1):    //ноль или элемент списка v:1
            [((v:1,0):<=,(v:1,0):>):?]}, //в зависимости от знака v:1
      //если длина списка > 1
/*сумма первых двух элементов списка и список v после удаления двух его
первых элементов или только список v с изъятой первой парой элементов
подаются на вход рекурсивной функции Add*/
      { ( ((v:1,v:2):+, v:-1:-1:[]):Add:. , (v:-1:-1:[]):Add:. ):
            [((v:1,0):>,(v:2,0):>):?]}// в зависимости от знаков v:1, v:2
  ) :. >>return;     }
```

Функция записана неправильно. Основная ошибка заключается в условии [((v:1,0):>,(v:2,0):>):?]. Вместо того чтобы суммировать положительные элементы, будет производиться команда ((v:1,v:2):+, v:-1:-1:[]):Add:., если первый элемент положительный (v:1,0):> и команда (v:-1:-1:[]):Add:., если второй элемент списка положительный (v:2,0):>. То есть если второй элемент отрицательный, то все равно находится сумма элементов и дополнительно, если оба элемента положительные, то выполнятся обе команды, произойдут два рекурсивных вызова.

Для отслеживания корректности вычислений программы добавлены пользовательские формулы к вершинам-операторам:

№24 – (NODE,0):>= - результат оператора больше либо равен нулю;

№20 – ((NODE:1,0):>, (NODE6,0):>, (NODE7,0):>):* – результат оператора список, первый элемент которого больше нуля и первый элемент списка v (NODE6 – это результат оператора с автоматически присвоенным номером 6, то есть v:1) больше нуля и второй элемент списка v больше нуля;

№28 – (NODE,0):>= - результат оператора больше либо равен нулю, здесь подразумевается, что формула должна быть истинной только на последней итерации, на других итерациях результатом будет выбор рекурсивного вызова функции Add.

Пусть спецификация начальных данных имеет вид (~ge0,~lt0). Тогда первая итерация рекурсии даст следующие результаты (рисунок 4.12). На рисунке 4.12 цветом выделены те вершины-операторы, которые были вычислены при заданном виде входных данных. В среде разработки, как для отладки, так и для верификации, можно включить или отключить поддержку задержанных вычислений, при ее отключении вычисляются все операторы функции, при включении задержанные операторы будут вычислены, если к ним обращается выполняемый оператор.

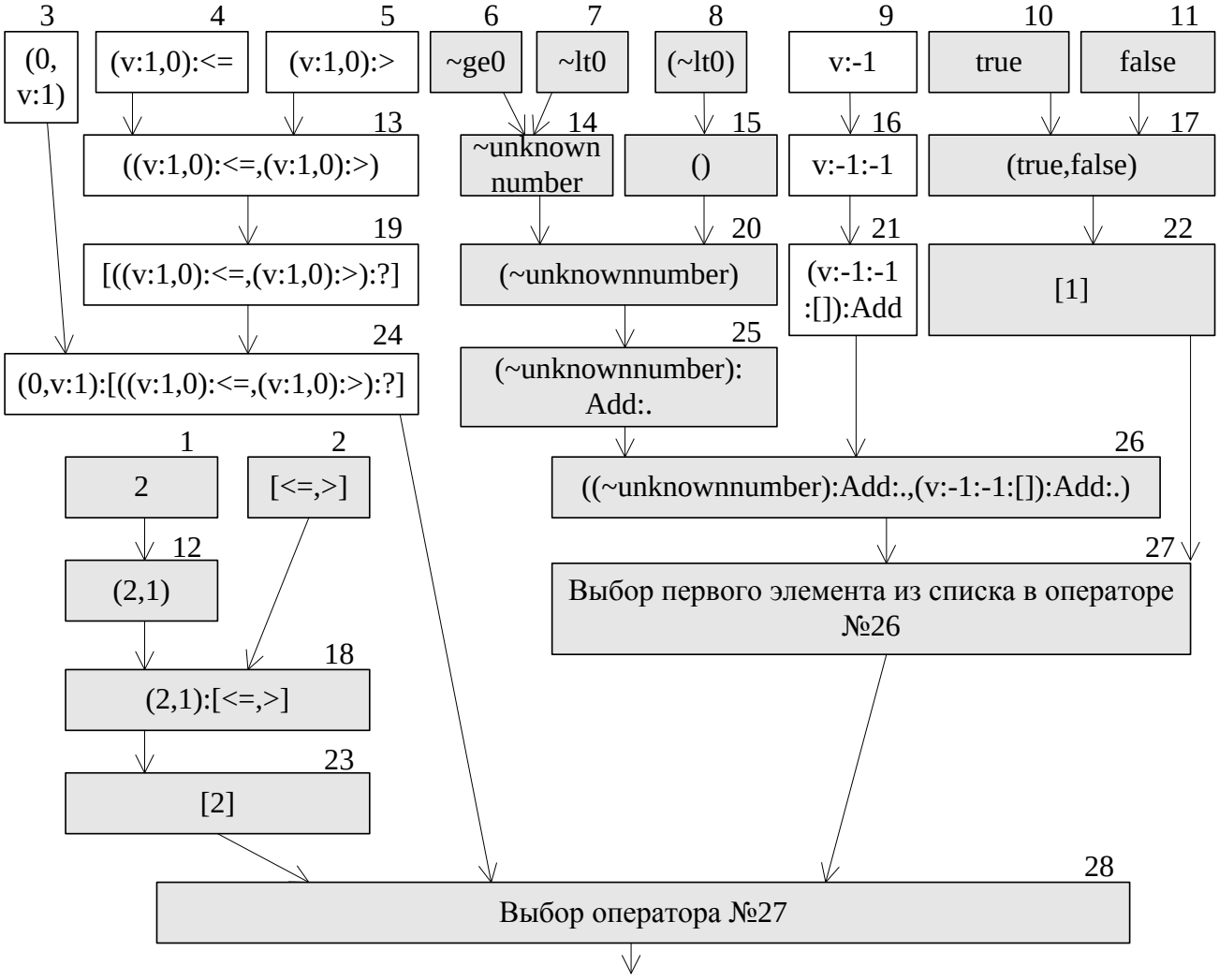


Рисунок 4.12 – Первая итерация функции Add над данными (~ge0,~lt0)

Пользовательские условия на графе при входных данных (~ge0,~lt0), примут следующие значения:

№24 – (NODE,0):>= - ~unknown, так как оператор #24 не был вычислен при указанных входных данных;

№28 – (NODE,0):>= - ~unknown, так как результат работы оператора #28 это вызов (~ge0):Add:.;

№20 – ((NODE:1,0):>, (NODE6,0):>, (NODE7,0):>):* – (true, true, false):* даст значение false, и оператор №20 будет помечен на графе как некорректный, то есть не соответствующий требованию пользователя.

Таким образом, пользовательская формула покажет наличие ошибки в программе. Необходимо заметить, что ошибочные команды могут содержаться не в самом операторе, к которому приписана формула, а в других вычисленных вершинах-операторах.

Пусть спецификация начальных данных имеет вид (~ge0,~ge0), первая итерация рекурсии даст такие результаты (рисунок 4.13).

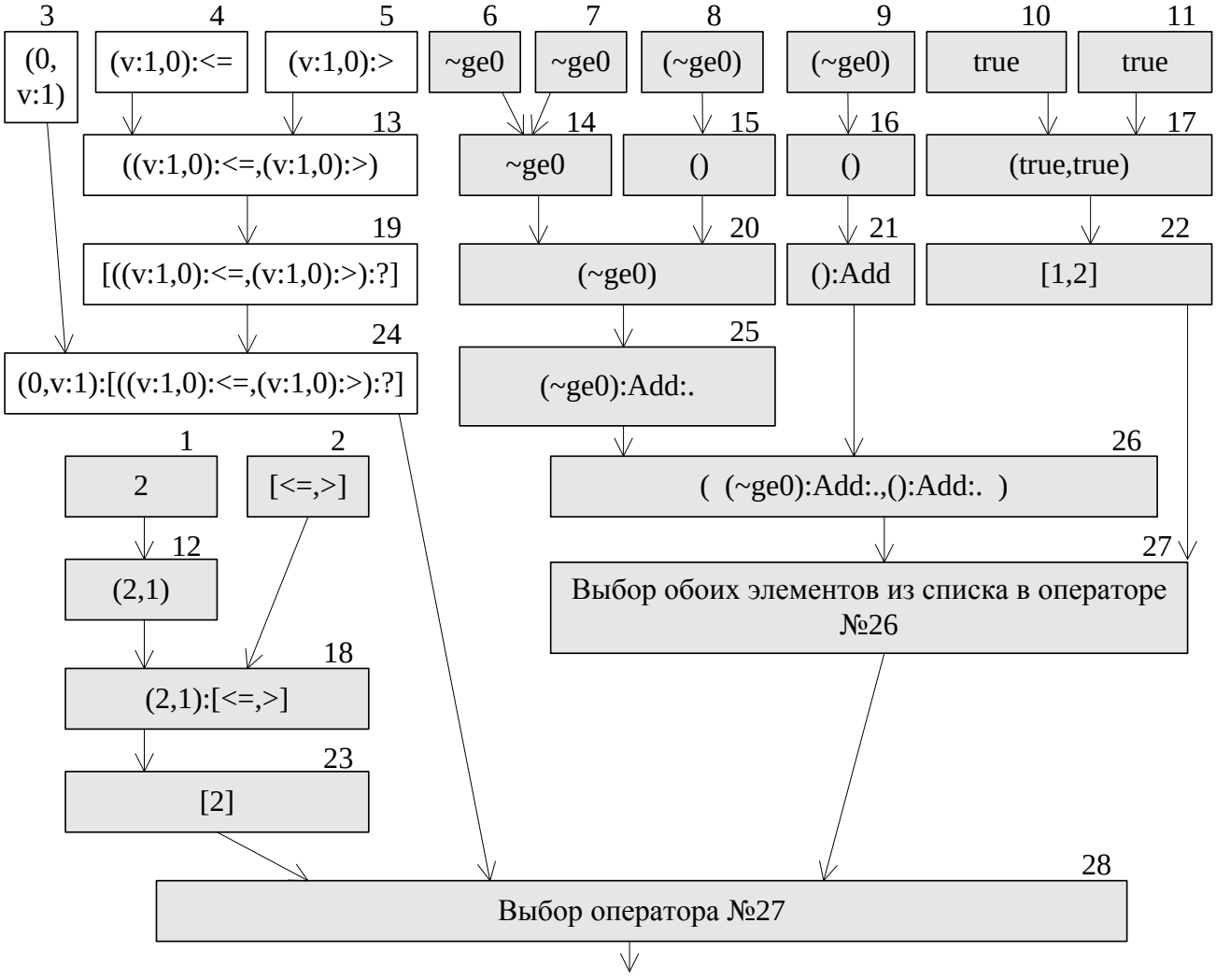


Рисунок 4.13 – Первая итерация функции Add над данными (~ge0,~ge0)

Пользовательская формула для оператора №20 в этом случае выдаст результат true, то есть не покажет наличия ошибки в функции. Однако на самом графе можно увидеть, что выполняются обе ветви, ведущие к рекурсивному вызову Add. Также при рассмотрении следующих итераций, будет видно, что произведутся вызовы функции Add над (~ge0) и (), вычисленные результаты будут равны ~ge0 и 0, и окончательный результат вычислений будет (~ge0, 0). Формула к результирующему оператору №28 в окончании вычислений будет иметь вид ((~ge0, 0), 0):> и даст ~unknown.

Условия (выражения), приписанные к вершинам-узлам графа, при определенных входных данных способны подтвердить или опровергнуть выполнение требований пользователя к результатам вычислений. Но если требования записаны неполно или ошибочно, включая как логические, так и синтаксические ошибки, то на уровне взаимодействия с пользователем-разработчиком программы, возможны ситуации, когда подтвержденная или опровергнутая корректность не имеет отношения к тому, что пользователь подразумевал под записью своих условий.

4.5 Выводы

Предложены и реализованы в рамках инструментальной среды для создания, отладки и верификации функционально-потоковых параллельных программ следующие возможности верификации.

1 Выделение в коде программы операторов и ветвей операторов, вычисляющих не используемые в дальнейшем значения, не требующее выполнения программы и спецификации пользователя.

2 Выявление задержанных последовательностей операторов, не имеющих шанса на выполнение ни при каких входных данных, не требующее выполнения программы и спецификации пользователя.

3 Проверка рекурсивных функций, не требующая выполнения программы и спецификации пользователя, определяющая ситуацию бесконечного

рекурсивного выполнения при любых входных данных, обусловленного неверным применением задержанных списков или их отсутствием.

4 Верификация функционально-потоковой параллельной программы с асинхронными списками, состоящая в переборе всех возможных комбинаций обработки асинхронного списка для каждой операции выбора элемента, позволяющая установить, существуют ли пути выполнения программы, приводящие к различным вычисленным значениям. Верификация программы с асинхронными списками не требует спецификации пользователя и происходит посредством вычисления операторов программы над конкретными входными данными.

5 Верификация программы со спецификацией пользователя, позволяющая описать входные данные программы в общем виде, рассматривающая автоматически операторы программы, получающие входные данные в общем виде, с помощью набора известных правил и запрашивающая мнение пользователя, если подходящего правила не нашлось. Дополнительно поддерживается возможность определения пользовательских условий для произвольных операторов программы.

Верификация со спецификацией пользователя может быть представлена пользователю несколькими способами, сходными с режимами проведения отладки. Модель функционально-потокового параллельного языка позволяет проводить верификацию на графе программы, повышая наглядность процесса верификации и позволяя сужать множество входных данных для каждого оператора.

Представленные возможности верификации функционально-потоковых параллельных программ проверяют корректность программы, как без участия пользователя, так и в режиме получения спецификации и запросов к пользователю.

5 Инструментальная поддержка отладки и верификации функционально-потоковых параллельных программ

Для практической апробации предложенных режимов отладки и верификации ФПП программ разработана инструментальная среда [17, Приложение В], предоставляющая редактор ФПП программ, процессы запуска трансляции и интерпретации, отладчик и верификатор. Создание инструментальной среды для разработки, отладки и верификации ФПП программ позволило на практике оценить особенности предложенных режимов отладки и верификации.

5.1 Структура среды разработки, отладки и верификации

Среда разработки является единым приложением с графическим интерфейсом, способным порождать процессы трансляции и интерпретации и обрабатывать результаты их выполнения. Но возможно выделить такие составляющие ее программные блоки.

1 Транслятор текста программы во внутренние представления, используемые отладчиком. Здесь будем называть его транслятором среды, так как приложение использует еще и внешний транслятор, а также интерпретатор ФПП программ, включенные в среду уже готовыми программными продуктами (рисунок 5.1).

2 Отладчик ФПП программ, получающий внутреннее представление программы и набор опций, регулирующих включение различных режимов отладки, выполняющий операторы программы с помощью запросов к интерпретатору. Порядок обхода операторов и форма взаимодействия с пользователем определяются опциями отладки.

3 Верификатор ФПП программ получает оператор и его входные значения, содержащие формулы спецификации (возможно наряду с точными значениями), осуществляет поиск результата работы оператора в наборе правил и возвращает

найденное значение или константу спецификации ~unknown, если поиск не дал результата. Запуск верификатора инициируется отладчиком при появлении формул спецификации во входных данных рассматриваемых операторов. Таким образом, верификация представляется пользователю в тех же режимах отображения, что и отладка, отличие состоит только в том, что отладка обрабатывает только точные значения, а верификация – утверждения спецификации.

4 Блок управления, координирующий совместную работу отладчика, верификатора, интерпретатора, внешнего транслятора и транслятора среды, и предоставляющий интерфейс пользователю для предоставления ему данных и получения от него запросов.

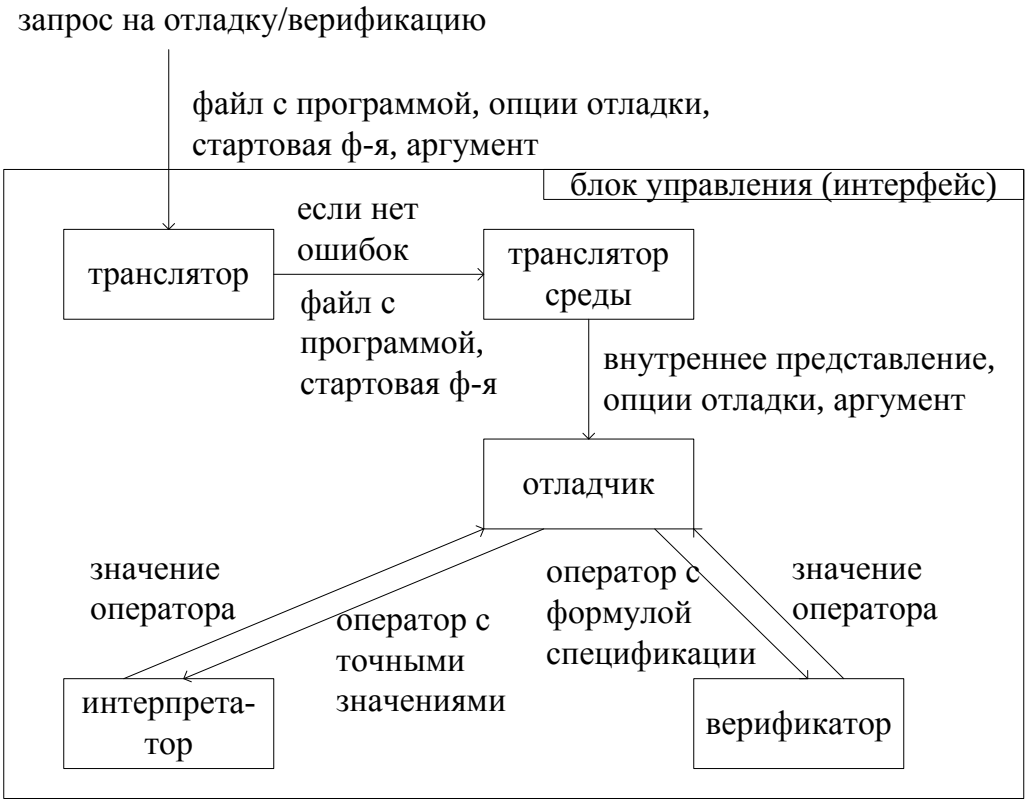


Рисунок 5.1 – Модули среды разработки, отладки и верификации ФПП программ

Программный код на языке C++ транслятора программы во внутренние представления, отладчика и элементы кода блока управления среды зарегистрированы как [34], программный код верификатора представлен в [17].

5.1.1 Транслятор

Транслятор (рисунок 5.2) обрабатывает файл с текстом программы и выделенной в нем стартовой функцией, создавая внутреннее представление программы, в дальнейшем используемое отладчиком.

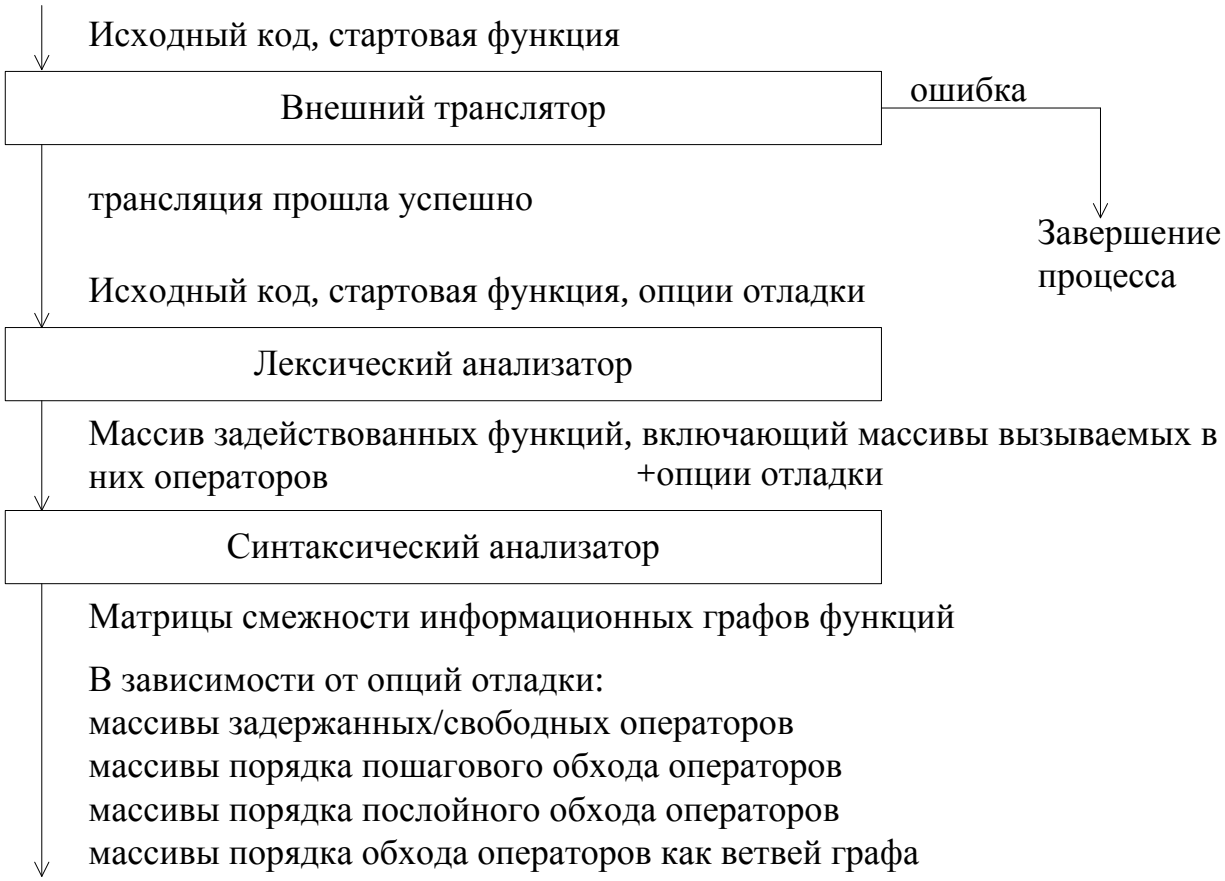


Рисунок 5.2 – Структура транслятора

Лексический анализатор выделяет границы записи кода стартовой функции, определяет множество функций из файла, которые могут быть вызваны при выполнении программы, начиная с указанной стартовой функции, выделяет их границы и составляет массив, содержащий описание каждой функции (рисунок 5.3).

Для каждой функции заполняются такие данные, как имя функции, код функции, аргумент функции, строки и столбцы начала и конца записи функции в тексте программы. Код функции и границы записи ее в тексте в дальнейшем могут понадобиться отладчику для внесения результатов вычислений операторов

в код функции, корректного отображения текста всей программы после отладочных подстановок и для выделения операторов в отлаживаемой функции.

Далее лексический анализатор в каждой выделенной функции считывает все описанные в их коде операторы интерпретации, группировки в список, копирования. В описании функции заполняются число операторов функции и динамический массив, содержащий описание каждого оператора (рисунок 5.3).

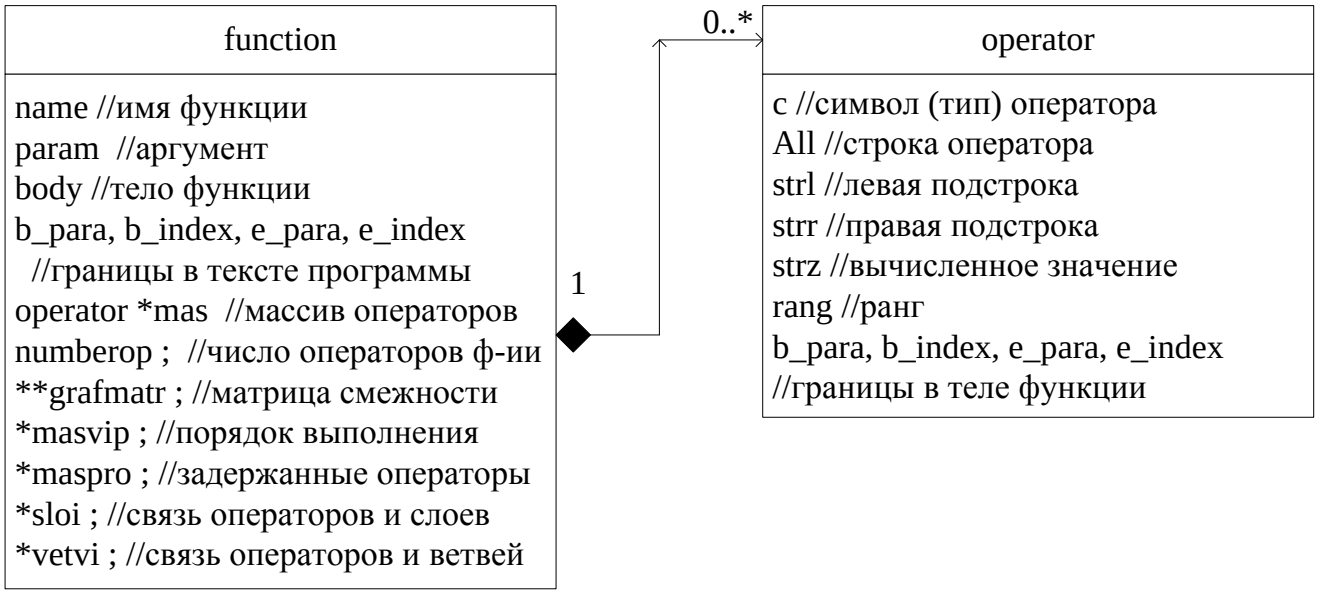


Рисунок 5.3 – Диаграмма структур внутреннего представления

Синтаксический анализатор на основе массивов операторов для каждой функции, сгенерированных лексическим анализатором, составляет матрицы смежности информационных графов функций, заполняет массивы, отображающие, являются ли операторы задержанными, номера слоев (ярусов) и ветвей операторов на информационном графе, номера операторов в соответствии с возможным последовательным обходом информационного графа функции. Синтаксический анализатор работает без учета поиска ошибок в записях операторов. Построенное представление программы не сохраняется в файле.

Операторы в теле функции выделяются через нахождение их обозначений в языке, для языка Пифагор это :, (), [], {}, >>, <<. Аргументы операторов определяются через подсчет открывающихся и закрывающихся скобок или достижения других специальных символов.

Матрица смежности является двумерным массивом, номера строк и столбцов соответствуют номерам операторов в массиве операторов функции. В строке для оператора отмечаются, значения каких других операторов требуется знать, чтобы начать вычисление оператора с номером, аналогичным номеру строки. Выделение операторов, чьи значения требуются текущему оператору, осуществляется через нахождение их кода или идентификаторов в тексте того оператора, чья строка заполняется.

При составлении порядка выполнения операторов и размещении их по слоям, в матрице смежности сначала выделяются все те операторы, чьи строки не содержат вхождений других операторов. Это значит, что все эти операторы могут быть выполнены в самом начале вычисления функции. Они в отдельном массиве sloi (рисунок 5.3) помечаются как операторы первого слоя, а также в произвольном порядке размещаются в массиве порядка выполнения masvip. Затем по матрице смежности отслеживаются все те операторы, которые нуждаются только в значениях операторов из первого слоя. Они помечаются в массиве sloi как операторы второго слоя, и добавляются в массив порядка выполнения masvip в произвольном порядке, но строго после операторов первого слоя. Формирование третьего и последующих слоев происходит аналогично.

Массив задержанных операторов формируется из тех операторов, чей код включен в задержанный список, чьи границы помечаются фигурными скобками в ФПП языке Пифагор.

При заполнении массива ветвей в матрице смежности находится первый оператор, не требующий значений других операторов. По матрице смежности отслеживается тот оператор, которому требуется значение предыдущего выделенного оператора. Если ему требуются значения и других операторов, то первая ветвь будет состоять только из одного начального оператора, а для начала выделения следующей ветви будет взят один из операторов, готовых к вычислению. В противном случае, оператор также добавляется к первой ветви и ищется следующий оператор, которому для вычисления достаточно только значения предыдущего оператора и т.д.

Описанный массив представления задействованных функций function (рисунок 5.3), содержащий массивы вызываемых в них операторов, является внутренним представлением программы для отладки и верификации.

5.1.2 Отладчик

Отладчик получает имя стартовой функции, ее аргумент, внутреннее представление программы от транслятора, опции, определяющие режимы проведения отладки и отображение ее пользователю. В режиме отладки «Проверка формул» дополнительно задается список пользовательских формул (условий), прикрепленных к операторам-вершинам стартовой функции. Задача отладчика – вычисление операторов программы и отображение результатов их работы в соответствии с выбранным режимом, взаимодействие с пользователем.

Вычисляемые операторы отладчик располагает в отдельном динамическом двусвязном списке, запоминая в его узлах значение структуры operator (рисунок 5.3), номер оператора в массиве операторов функции, номер функции в списке функций и флаг, показывающий вычислялся ли оператор. Кроме списка операторов отладчик создает список вызываемых во время отладки функций, запоминая код функции, ее номер в массиве функций, построенном транслятором, и уровень вложенности рекурсивного вызова. Список функций необходим при наличии в программе рекурсии, так как на каждой новой итерации надо заново корректировать код функции для отображения пользователю и по ее завершению возвращаться к коду функции на предыдущей итерации.

Отладчик генерирует список операторов в зависимости от указанного режима. Для режима пошаговой отладки последовательность операторов совпадает с описанной транслятором в masvip (рисунок 5.3), для режимов отладки слоев и отладки ветвей при составлении списка задействуются еще и данные из sloi и vetvi. Сначала в список добавляются операторы только стартовой функции. Отладчик выполняет по одному оператору за шаг отладки в режиме пошаговой отладки и по набору операторов в режимах отладки слоев и отладки ветвей, число

операторов в наборе определяется с помощью массивов sloi и vetvi. Результаты вычисления операторов предоставляются пользователю в текстовом и графическом виде (если последний включен пользователем в опции отладки), далее пользователь дает либо команду произвести следующий шаг отладки, либо команду завершить отладку. В последнем случае очищается память от внутреннего представления программы и списков отладчика, а на уровне интерфейса пользователь возвращается в режим редактирования файла с программой.

В список операторов задержанные операторы помещаются наряду со всеми остальными. Но если в опциях отладчика указано выполнять программу, учитывая задержанные списки, то на шаге отладки задержанный оператор пропускается (оставаясь в списке операторов как невычисленный), а при вычислении обычного оператора, идет определение того, существует ли запрос на данные из задержанного списка. Если такой запрос существует, идет обратный рекурсивный обход списка операторов и с учетом матрицы смежности, выбирается вся требуемая ветвь операторов, которая и будет вычислена за один шаг отладки, даже если активирован режим пошаговой отладки.

При обработке отладчиком списка операторов возможны и такие ситуации, как появление оператора вызова функции и вычисление оператора интерпретации над параллельным списком. При выполнении оператора, вызывающего функцию, сразу производится полное вычисление результата и его отображение на шаге отладки. Но если в опциях отладки включен флаг «Запуск других функций», то следующий шаг отладки дополнит список операторов командами вычисленной функции и начнет их пошаговое выполнение вместе с предоставлением данных пользователю уже по информационному графу и программному коду новой функции. По завершению рассмотрения всех команд вызванной функции (а возможно и команд других вызываемых ею функций) следующий шаг отладки вернется к командам начальной функции. При выполнении оператора интерпретации над параллельным списком аналогичным образом произойдет вычисление всех команд в полном объеме и отображение их результата, но

следующие шаги отладки начнут рассмотрение вычислений для каждого элемента параллельного списка в отдельности.

Отладка в режиме «Проверка формул» основана на тех же принципах и выполняется тем же программным модулем, отличия состоят в шаге отладки, вычисляющим все операторы одной функции и в обработке рассматриваемых операторов. Отладчик, выполняя оператор, определяет, существуют ли приписанные к нему пользовательские формулы (условия) и вычисляет их, подставляя в формулу требуемые значения вершин-операторов или аргумента функции. При этом проверка синтаксиса записи пользовательских условий не производится, если они не верны, то результатом будет ошибка. Если условие пользователя задействует значение еще не обработанного оператора, то значение условия также будет ошибочным.

Для вычисления результата работы оператора отладчик по матрице смежности отслеживает значения его аргументов. Если среди аргументов есть хотя бы одна формула спецификации, то оператор подается на вход верификатору, в противном случае на вход интерпретатору. Такой же принцип применяется и к пользовательским условиям. Полученное от интерпретатора или верификатора значение подставляется в значение оператора или пользовательской формулы. Исключение составляет получение от верификатора константы спецификации ~unknown, при этом производится запрос к пользователю о результате работы оператора. Ответ пользователя заносится в значение оператора или условия без проверки на корректность.

5.1.3 Верификатор

Верификатор получает оператор и его аргументы. Если это оператор группировки в список или оператор копирования, то осуществляется простая подстановка значений. Если это оператор интерпретации и среди аргументов есть параллельный список, производится разбиение выражения на несколько составляющих его операторов. В общем случае выделяется функция, указанная

для интерпретации, и если это операция языка программирования Пифагор, то вызывается отдельная функция в программном коде верификатора для ее обработки. Для оператора интерпретации функции, введенной программистом, верификатор выдаст ответ ~unknown, но более детальное рассмотрение действий функции над аргументом, заданным спецификацией пользователя, может быть доступно на следующих шагах верификации (отладки).

Верификатор оперирует функциями обработки операций языка Пифагор [50, 57, 58], это: |, ., (), [],{}, (.), ?, #, dup, %, .., целое число, +, -, *, /, =, !=, >, <, >=, <=. Аргументы операций, обрабатываемых верификатором, могут быть не только выражениями спецификациями, но и точными значениями, однако в этом случае результаты над ними будут вычисляться с помощью команд языка C++, что может не совпасть со значениями, вычисленными интерпретатором. Для оператора с аргументами, заданными выражениями спецификации, функция верификатора осуществляет вычисление результата по специальным правилам, основную часть которых можно увидеть в приложении Б. При отсутствии правила формируется ответ ~unknown.

Большая часть выражений спецификации описывает интервалы. Для их обработки в коде верификатора введена специальная структура с описанием арифметических и логических операций над ними по правилам, взятым из [5]. Рассматриваемый верификатором оператор может иметь смешанные аргументы, то есть часть из них могут быть заданы спецификацией, а часть точными значениями. В этом случае численные значения также рассматриваются как интервалы. Значение оператора, найденное верификатором, может быть как формулой спецификации, так и точным значением.

5.1.4 Блок управления

Блок управления определяет интерфейс взаимодействия с пользователем (рисунок 5.4), осуществляет запуск модулей среды и передачу данных между ними (рисунок 5.1).

Пользователь начинает работу с создания нового или открытия существующего проекта. Проект представляет совокупность файлов с кодом программ на ФПП языке, а также ряда файлов других форматов, например, таких как файлы промежуточного представления или файлы изображений с информационными графами функций. Но, несмотря на поддержку концепции проектов, и внешний транслятор, и транслятор среды обрабатывают только один отдельный файл с программой.

Рисунок 5.4 – Диаграмма запросов пользователя к интерфейсу блока управления

Выбрав файл из проекта, или добавив в него новый файл, пользователь получает доступ к редактированию, трансляции, интерпретации, выбору стартовой функции, построению дерева функций (отображается в окне интерфейса) и графа функции (сохраняется как bmp файл в папке проекта), отладке и верификации в различных режимах.

5.2 Интегрированная среда разработки, отладки и верификации функционально-потоковых параллельных программ

Интегрированная среда разработки, отладки и верификации ФПП программ предоставляет пользователю графический интерфейс для управления программными файлами, трансляции, интерпретации, отладки и верификации программ и осуществляет координацию вышеописанных программных модулей.

Главное окно среды разработки (рисунок 5.5) предоставляет текстовый редактор с подсветкой синтаксиса, список файлов проекта, список функций, записанных в выбранном программном файле, панель инструментов и меню.

Такие пункты главного меню, как Файл, Редактор, Настройки, Проект, Справка предоставляют преимущественно стандартные функции, например такие как, открытие и сохранение файлов или проектов, добавление и удаление файлов из проекта, функции работы с текстом, настройка оформления графической оболочки и командных строк транслятора и интерпретатора, справка о среде и особенностях языка.

В пункте меню Программа (рисунок 5.6) заключены все основные функции отладки и верификации.

Пункт меню Отладка дает доступ к проведению отладки (рисунок 5.7) в одном из трех режимов: пошаговой отладки, отладки ветвей и отладки слоев, а также к спецификации входного аргумента функции и последующего проведения формальной верификации функции с аргументом, заданном в общем виде. Обход функции и вид предоставляемой информации при верификации совпадает с выбранным режимом отладки.

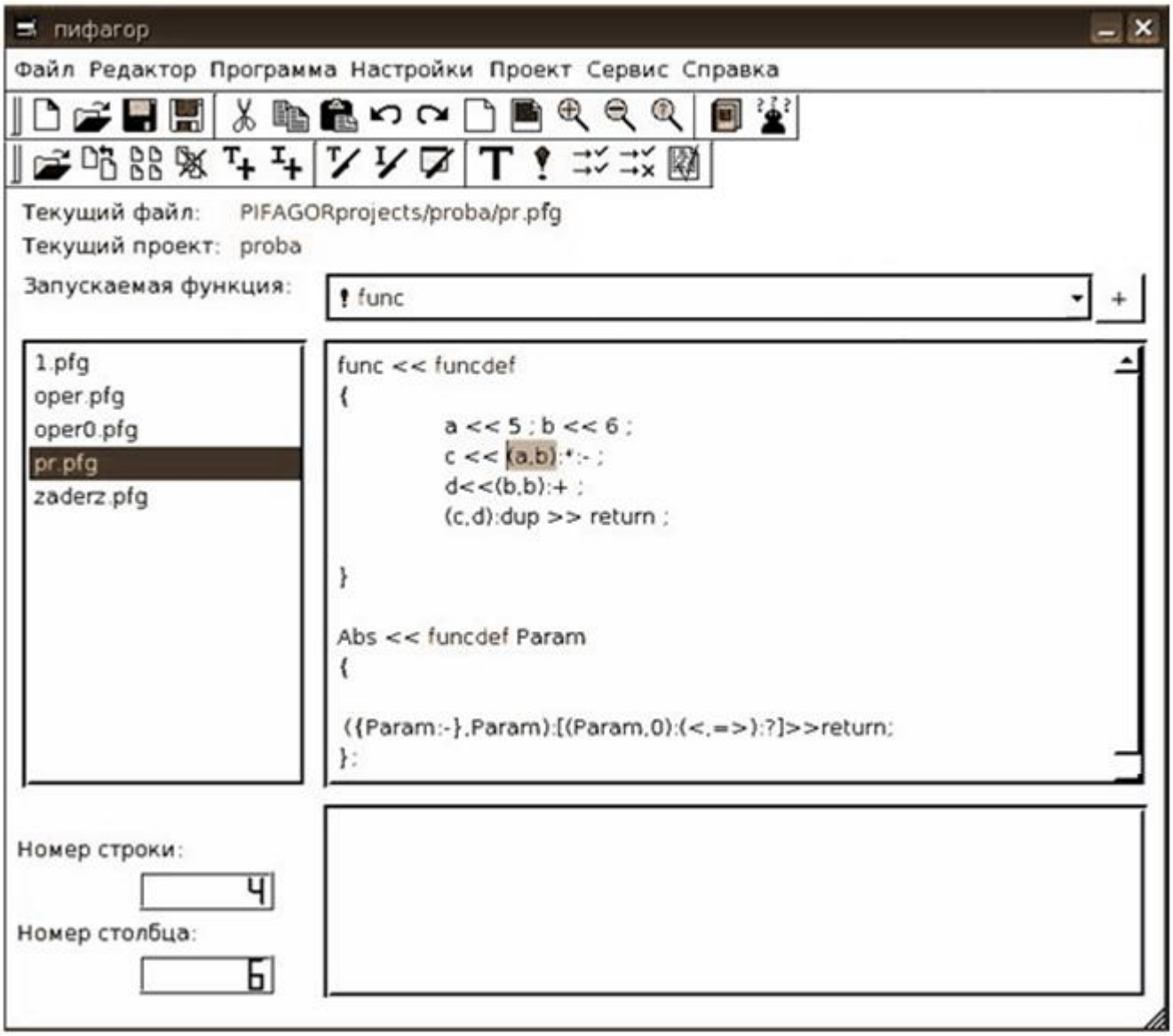


Рисунок 5.5 – Главное окно среды разработки

Рисунок 5.6 – Пункт меню Программа

Проводится ли отладка или верификация, будет зависеть от отсутствия или наличия в аргументе функции константы спецификации. Если аргумент задан

определенными значениями, проводится отладка, если присутствуют обобщенные данные – верификация.

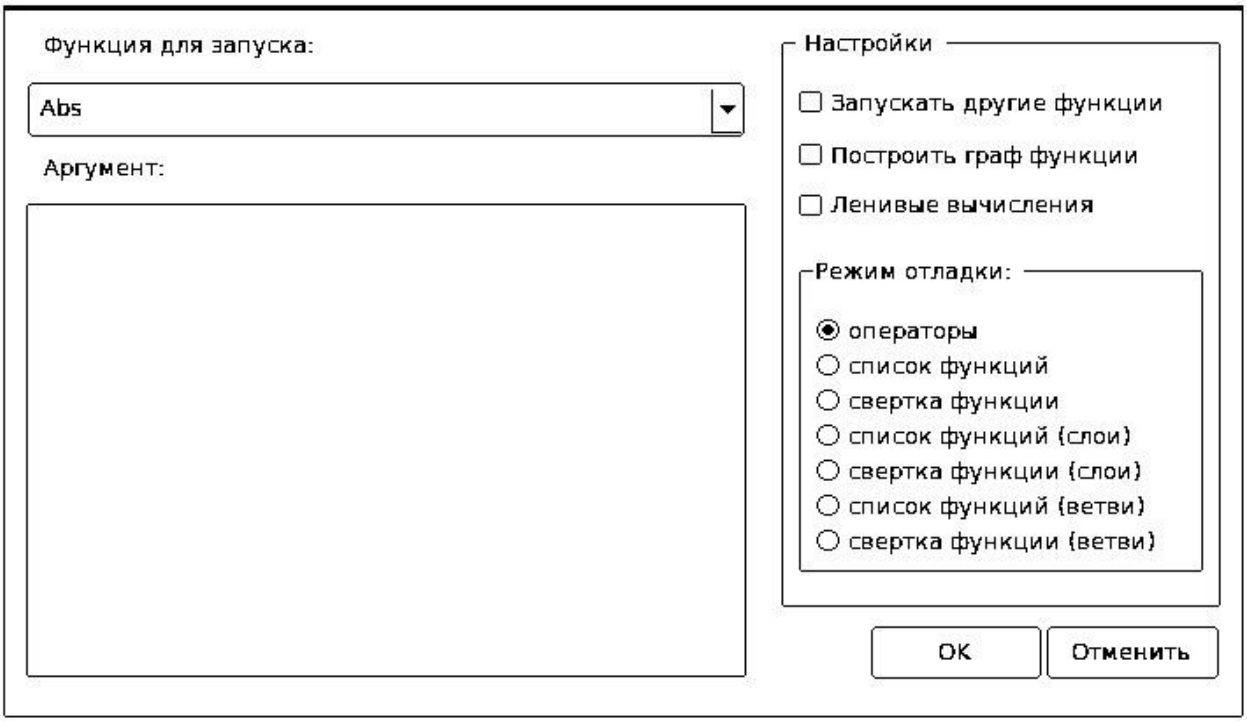


Рисунок 5.7 – Диалог выбора режима отладки или верификации

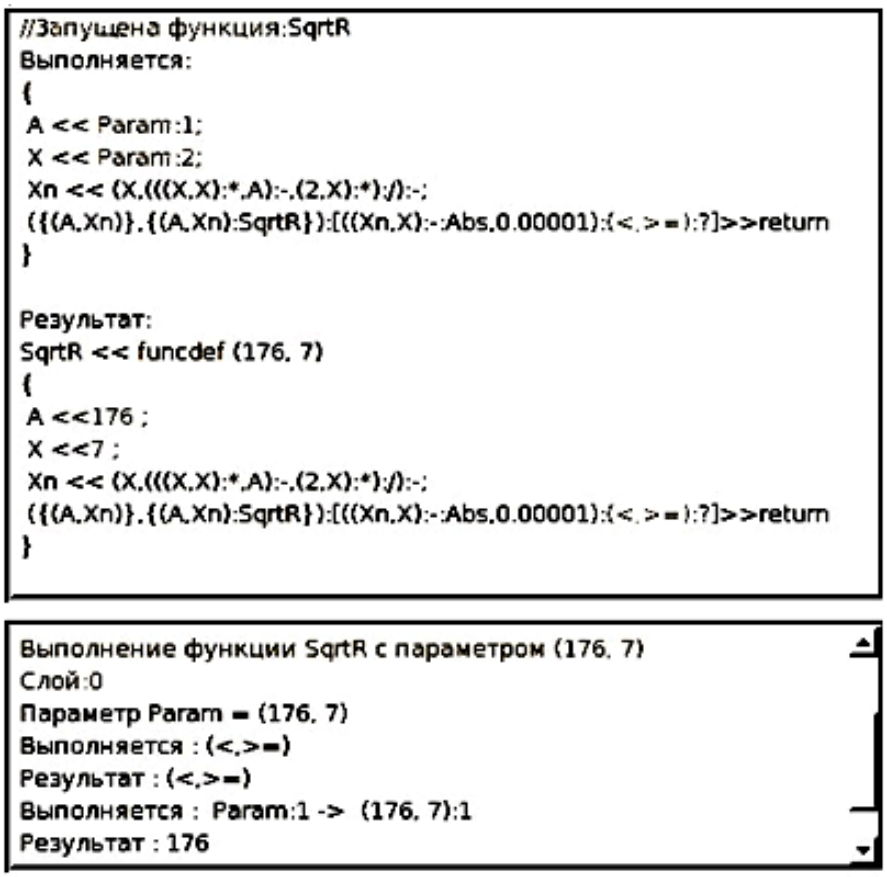


Рисунок 5.8 – Отладка в главном окне среды разработки

Процесс отладки отображается в главном окне среды разработки (рисунок 5.8). При этом нижнее поле вывода текста детально отображает каждый шаг отладки, все операторы, результаты подстановки в них уже вычисленных значений и результаты их выполнения. Содержимое, отображаемое в верхнем поле вывода текста, может различаться в разных режимах, но, как правило, там присутствуют текст функции до вычисления шага отладки и текст функции после вычисления шага отладки. Так, на рисунке 5.8 видно, что операторы шага отладки в режиме отладки слоев Param:1 и Param:2 были замещены на свои результаты, то есть на 176 и 7.

Если пользователь при запуске отладки или верификации выбрал строительство графа функции, то в правом верхнем углу окна появляется кнопка, позволяющая переключаться между текстовым (рисунок 5.8) и графическим (рисунки 5.9, 5.10) представлением процесса отладки или верификации.

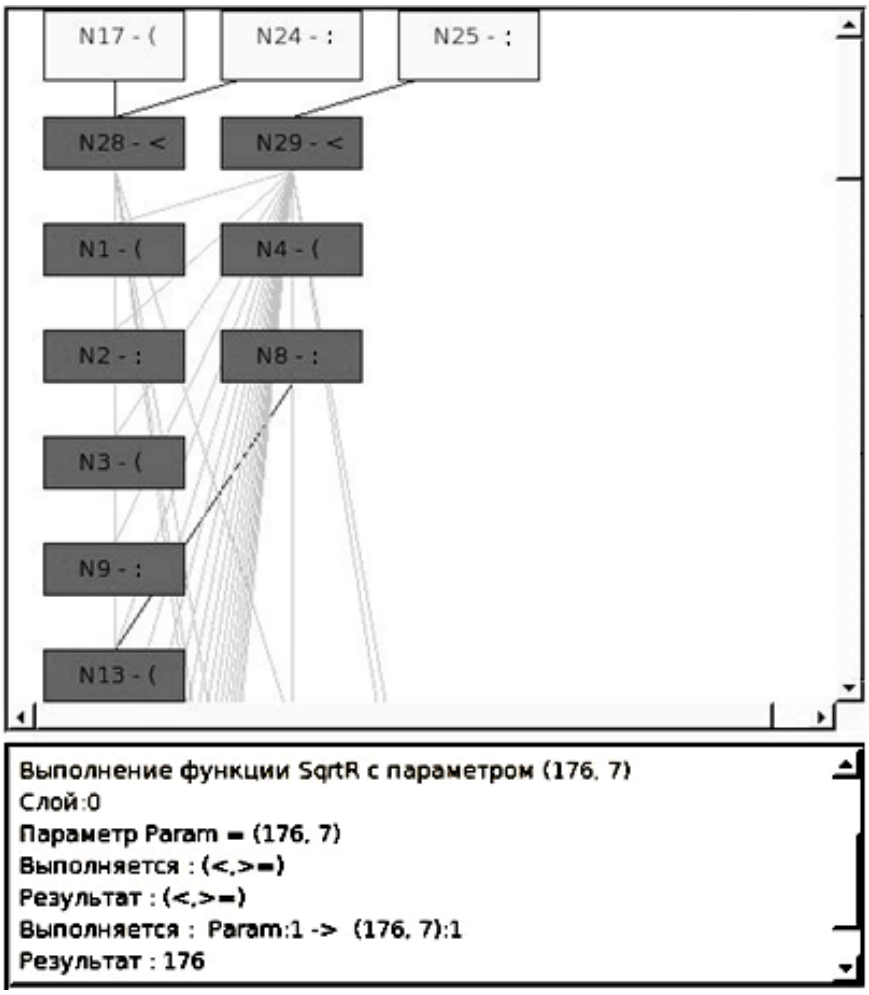


Рисунок 5.9 – Графическое представление отладки

Граф функции (рисунки 5.9, 5.10) позволяет при нажатии на любую вершину отобразить информацию о записи оператора в программе и результате работы оператора, если тот уже вычислен.

Как видно из рисунков 5.9 и 5.10 процесс верификации отличается от процесса отладки, только тем, что он обрабатывает данные, заданные в общем виде через константы спецификации (на рисунке 5.10 - ~lt-2.5).

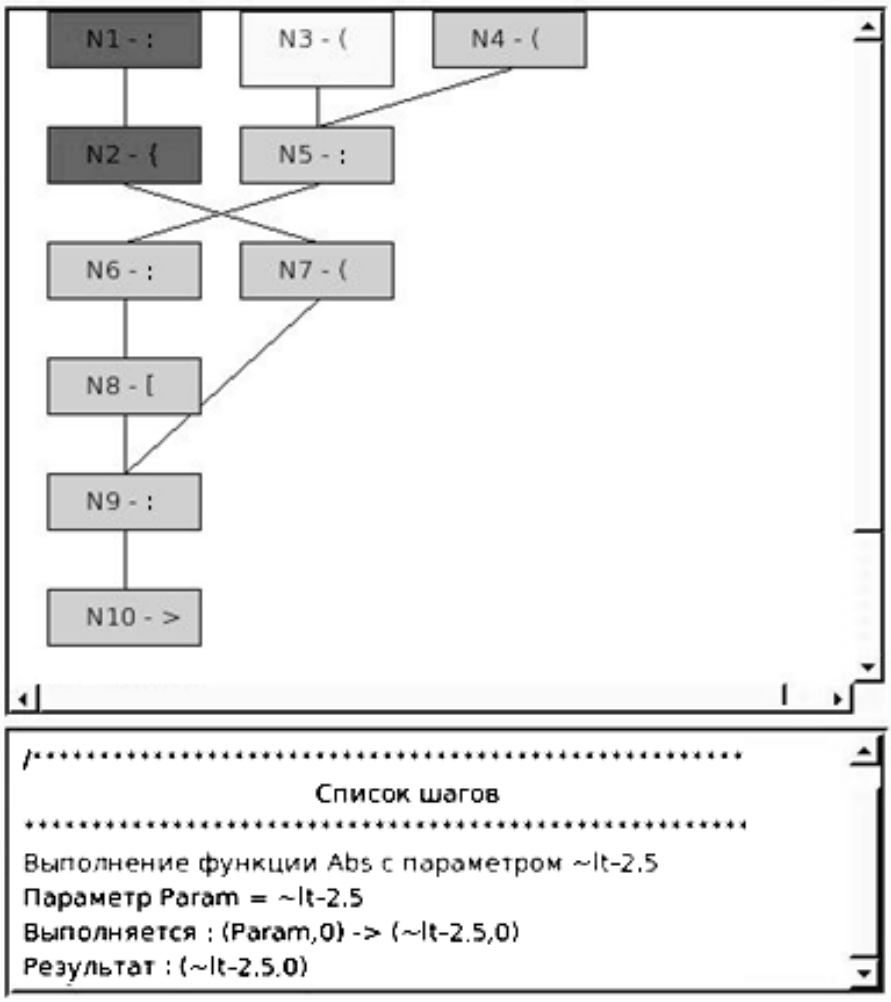


Рисунок 5.10 – Графическое представление верификации

Пункт меню «Проверка формул» (рисунок 5.6) запускает отладку или верификацию в четвертом режиме, в режиме проверки формул. Здесь пользователь вводит (рисунок 5.11) только имя функции, с которой начнется выполнения программы и ее аргумент.

После этого пользователю предоставляется граф функции (рисунок 5.12). По нажатию на любую вершину графа появляется диалог (рисунок 5.13), позволяющий ввести множество формул, которые будут приписаны к указанной

вершине. Далее шаг отладки совпадает с итерацией функции (для не рекурсивных функций обработка будет завершена за один шаг). На шаге отладки вычисляется множество вершин-операторов и дополнительно подстановкой вычисляются все пользовательские формулы.

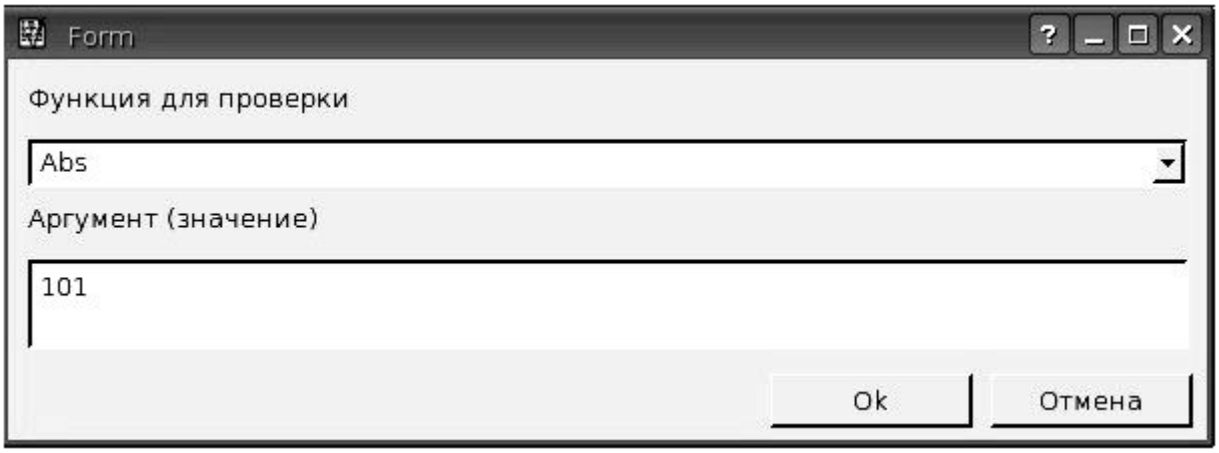


Рисунок 5.11 – Диалог начала отладки в режиме проверки формул

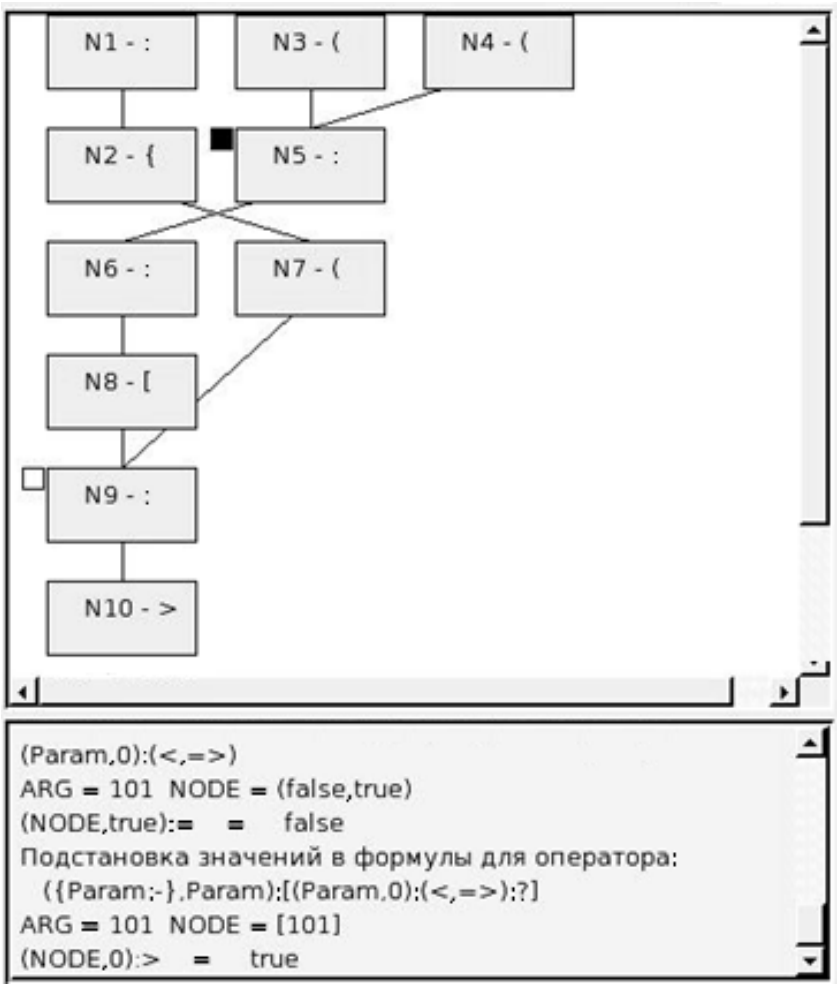


Рисунок 5.12 – Отладка в режиме проверки формул

Если все формулы, указанные пользователем для вершины, возвращают истину (true), то вершина помечается корректной и размечается цветом. Если хотя бы одна из формул возвращает значение отличное от истины, то вершина помечается не корректной (рисунок 5.12). Так, на рисунке 5.12, вершина №9 помечена корректной, а вершина №5 не корректной, для остальных вершин пользовательские формулы не были заданы.

Диалог задания формул (рисунок 5.13) позволяет ввести множество формул, приписанных к узлу графа, а также показывает соответственный узлу графа оператор в тексте программы, его вычисленное значение, вычисленные значения пользовательских формул и отметку об истинности или ложности формулы. Данные диалога будут доступны сразу после проведения шага отладки, по нажатию на вершину графа.

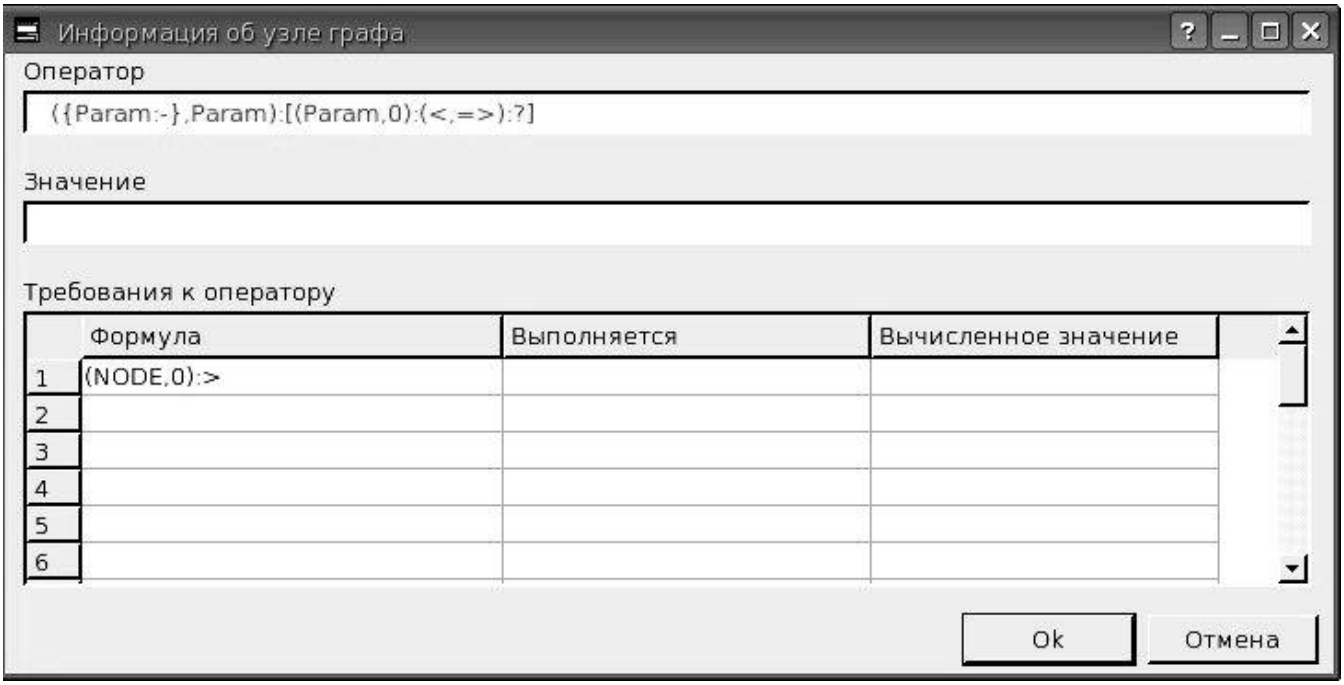


Рисунок 5.13 – Диалог задания формул в режиме проверки формул

Пункты меню Проверка программы и Проверка асинхронных списков (рисунок 5.6) отображают диалог, аналогичный представленному на рисунке 5.11. Если была выбрана проверка программы, то пользователь вводит только имя интересующей его функции, затем происходит проверка матрицы смежности графа функции на наличие вершин, не связанных с другими вершинами и

отличными от выхода из функции или завершения блока. Найденные вершины-операторы предоставляются пользователю.

Если была выбрана проверка асинхронных списков, то пользователь вводит имя функции и ее аргумент, после чего происходит выполнение выбранной функции. При достижении оператора, связанного с обработкой асинхронного списка, происходит вычисление всех возможных вариантов его исполнения, учитываются и варианты исполнения предыдущих операций с асинхронными списками. Таким образом, устанавливаются все возможные результаты, вычисляемые функцией с асинхронным списком, и предоставляются пользователю.

Пункт меню Сервис (рисунок 5.14) позволяет построить список функций и дерево функций (рисунок 5.15) для выбранного программного файла, используя список и дерево, пользователь может переместиться к выбранной функции в тексте программы.

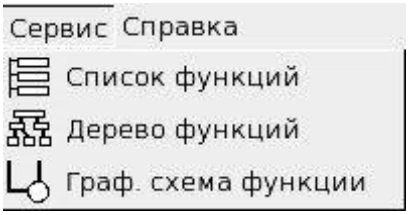


Рисунок 5.14 – Пункт меню Сервис

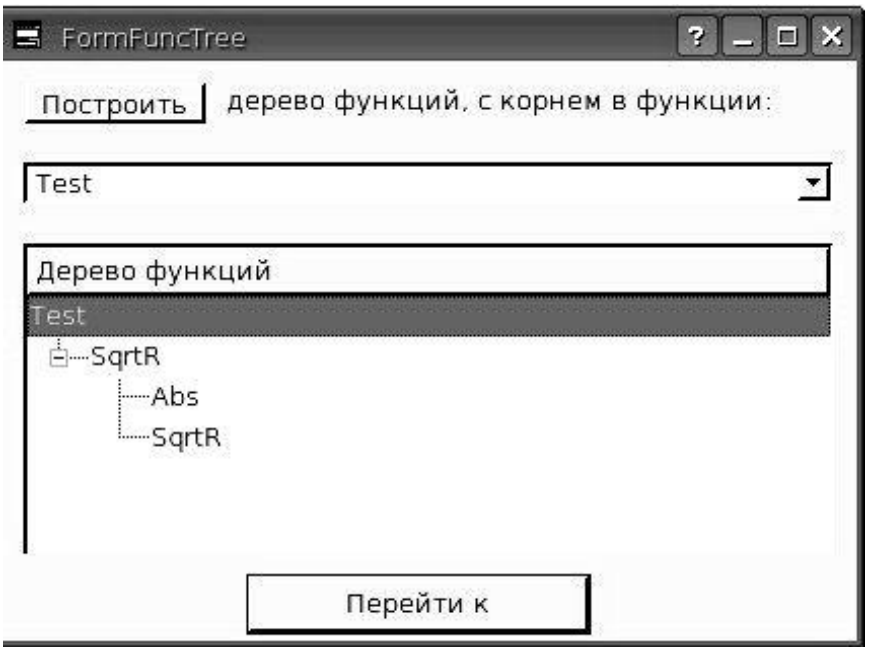


Рисунок 5.15 – Диалог дерева функций

При выборе построения графической схемы функции, будут созданы файлы формата bmp, отображающие граф выбранной функции. Пример автоматически построенного информационного графа на рисунке 5.16.

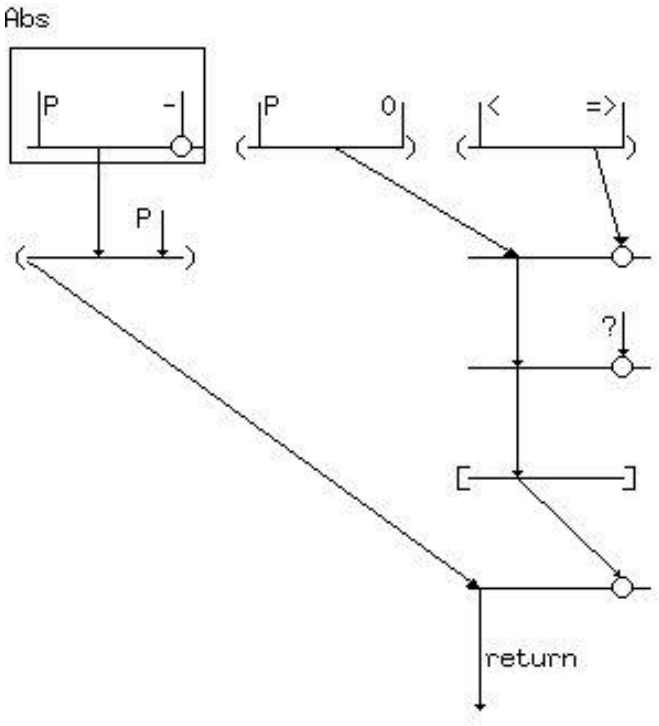


Рисунок 5.16 – Графическая схема функции

5.3 Выводы

Разработанная инструментальная среда для отладки и верификации функционально-потоковых параллельных программ, объединяющая средства графического интерфейса, транслятор во внутреннее представление, отладчик и верификатор, позволила реализовать для практического использования, предложенные в главах 3 и 4, методы отладки и верификации ФПП программ, а также исследовать особенности восприятия различных режимов отладки и верификации ФПП программ разработчиком.

Заключение

В рамках работы получены следующие научные и практические результаты.

1 Предложены и реализованы методы отладки ФПП программ, использующие различные формы обхода информационного графа программы с возможностью задания дополнительных условных выражений для проверки данных. Методы позволяют проводить детальный анализ процесса выполнения программы, уменьшать число шагов отладки, увеличивать эффективность локализации логических ошибок, отображать уровень параллелизма функций и синхронизацию ее операторов.

2 Разработаны способы применения методов индуктивных утверждений и индукции для функционально-потоковой модели параллельных вычислений с использованием информационного графа программы. На основе этого предложен метод формальной верификации функционально-потоковых параллельных программ, основанный на спецификации входных данных с использованием как конкретных значений, так и дополнительных условий, накладывающих ограничения на аргументы и промежуточные результаты. Это обеспечивает автоматизированную проверку программы для обобщенного множества входных данных и позволяет установить соответствие вычислений, выполняемых программой, спецификациям разработчика. Показано, что в ряде случаев возможно автоматическое определение свойства корректности программы или получение интервальной оценки вычисляемых числовых значений.

3 Предложен метод верификации функционально-потоковых параллельных программ, позволяющий установить, приводит ли произвольное появление данных внутри асинхронных списков к различным результатам вычислений.

4 Реализована среда разработки, обеспечивающая отладку и верификацию функционально-потоковых параллельных программ с использованием предложенных методов.

Список литературы

1. Абрамов, С.А. Элементы анализа программ / С.А. Абрамов. – М.: Наука, 1986. - 128 с.
2. Авербух, В. Л. Разработка средств визуализации программного обеспечения параллельных вычислений. Визуальное программирование и визуальная отладка параллельных программ / В. Л. Авербух, А. Ю. Байдалин // Вопросы атомной науки и техники. Математическое моделирование физических процессов. - 2003. - N 4. - С. 68-80.
3. Алагич, С. Проектирование корректных структурированных программ / С. Алагич, М. Арбиб. - М: Радио и связь, 1984. - 254 с.
4. Александров, В.В. Автоматизированная обработка информации на языке предикатов / В.В. Александров. - М.: Наука, 1982. - 103 с.
5. Алефельд, Г. Интервальный анализ: теория и приложения [Электронный ресурс] / Г. Алефельд, Г. Майер // Режим доступа: www.sbras.ru/interval/Introduction/ISurveyRus.pdf
6. Антонов, А.С. Введение в параллельные вычисления / А.С. Антонов. - М.: Изд-во МГУ, 2002. - 69 с.
7. Антонов, А.С. Параллельное программирование с использованием технологии MPI / А.С. Антонов. - М.: Изд-во МГУ, 2004. - 71 с.
8. Бакалов, Ю.В. Язык спецификации поведения распределенных программ / Ю.В. Бакалов, Р.Л. Смелянский // Программирование. - 1996. - N 5. - С.41-51.
9. Баканов, В.М. Введение в практику разработки параллельных программ в стандарте MPI / В.М. Баканов, Д.В. Осипов. - М.: МГАПИ, 2005. - 63 с.
10. Барский, А.Б. Потоковая вычислительная система: программирование и оценка эффективности / А.Б. Барский, В.В. Шилов // Информационные технологии. - 2003. - N 7. - С. 2-23.
11. Батищев, Д.И. Методы оптимального проектирования. / Д.И. Батищев. - М: Радио и связь, 1984. - 248 с.

12. Бурдонов, И.Б. Применение конечных автоматов для тестирования программ / И. Бурдонов, А. Косачев, В. Кулямин // Программирование. - 2000. - N 2. - С. 61-73.
13. Бурдонов, И.Б. Формальные спецификации в технологиях обратной инженерии и верификации программ / И.Б.Бурдонов, А.В.Демаков, А.С.Косачев, А.В.Максимов, А.К.Петренко // Труды Института системного программирования РАН. - 1999. - N 1. - С. 31-43.
14. Бурцев, В.С. Вычислительные процессы с массовым параллелизмом / В.С. Бурцев // Электроника: НТБ. - 2002. - N 2. - С. 32-35.
15. Васильева, К.А. Верификация автоматных программ с использованием LTL / К.А. Васильева, Е.В. Кузьмин // Моделирование и анализ операционных систем. - 2007. - N 1. - С. 3-14.
16. Вельдер, С.Э. О верификации простых автоматных программ на основе метода MODEL CHECKING / С.Э. Вельдер, А.А. Шалыто // Информационно-управляющие системы. - 2007. - N 3. - С. 27-38.
17. Верификатор функционально-потоковых параллельных программ на языке Пифагор под ОС Linux [Текст] : пат. 2013660557 Рос. Федерация / А.И. Легалов, Ю.В. Удалова ; заявитель и патентообладатель Федеральное государственное учреждение высшего профессионального образования "Сибирский федеральный университет" (СФУ) ; заявл. 18.11.2013 ; опубл. 15.01.2014
18. Верификатор Bogor [Электронный ресурс] Режим доступа: http://bogor.projects.cis.ksu.edu
19. Верификатор Isabelle [Электронный ресурс] Режим доступа: http://www.cl.cam.ac.uk/research/hvg/Isabelle
20. Верификатор PVS [Электронный ресурс] Режим доступа: http://pvs.csl.sri.com
21. Верификатор Relacy Race Detector [Электронный ресурс] Режим доступа: http://groups.google.com/group/relacy

22. Верификатор Spin [Электронный ресурс] Режим доступа: http://spinroot.com
23. Верификатор VCC [Электронный ресурс] - Режим доступа: http://vcc.codeplex.com
24. Визуальные средства создания, отладки и анализа программ для параллельных вычислений [Электронный ресурс] Режим доступа: http://www.itlab.unn.ru/file.php?id=332
25. Виноградов, Р.А. Верификация автоматных программ средствами CPN / Р.А. Виноградов, Е.В. Кузьмин, В.А. Соколов // Моделирование и анализ операционных систем. - 2006. - N 2. - С. 4-15.
26. Воеводин, В.В. Математические модели и методы в параллельных процессах / В.В. Воеводин. - М.: Наука, 1986. – 296 с.
27. Вудкок, Дж. Первые шаги к решению проблемы верификации программ // Открытые системы. - 2006. - N 8. - С. 35-57.
28. Гаврилов, Г.П. Логический подход к искусственному интеллекту: от классической логики к логическому программированию / Г.П. Гаврилов. - М.: Мир, 1990. - 429 с.
29. Гайсарян, С.С. О некоторых задачах анализа и трансформации программ / С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев, О.Р. Маликов, Д.М. Мельник, А.В. Меньшикова // Труды Института системного программирования РАН. - 2004. - N 5. - С. 7-41.
30. Гергель, В.П. Основы параллельных вычислений для многопроцессорных вычислительных систем. / В.П. Гергель, Р.Г. Стронгин. - Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003. – 184 с.
31. Глуханков, М.П. Транслирующие компоненты системы функционального программирования SFP [Электронный ресурс] / М.П. Глуханков, П.А. Дортман, А.А. Павлов, А.П. Стасенко // Режим доступа:http://www.iis.nsk.su/files/articles/sbor_kas_09_gluhankov_etc.pdf
32. Гордеев, Д.С. Визуализация внутреннего представления программ в системе функционального программирования SFP [Электронный ресурс] /

Д.С. Гордеев // Режим доступа: http://www.iis.nsk.su/files/articles/sbor_kas_16_gordeev.pdf

33. Городняя, Л.В. Основы функционального программирования / Л.В. Городняя. - М.: ИНТУИТ, 2004. – 280 с.
34. Графический отладчик функционально-потоковых параллельных программ на языке Пифагор под ОС Linux [Текст] : пат. 2012612190 Рос. Федерация / А.И. Легалов, О.В. Непомнящий, Ю.В. Удалова, В.А. Хабаров ; заявитель и патентообладатель Федеральное государственное учреждение высшего профессионального образования "Сибирский федеральный университет" (СФУ) ; заявл. 28.12.2011 ; опубл. 28.02.2012
35. Грис, Д. Наука программирования / Д. Грис. - М.: Мир, 1984. - 416 с.
36. Гуров, В.С. Технология верификации автоматных моделей программ без их трансляции во входной язык верификатора. / В.С. Гуров, А.А. Шалыто, Б.Р. Яминов // Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы». - 2007. - С. 198-203.
37. Дейкстра, Э. Дисциплина программирования / Э. Дейкстра. - М.: Мир, 1978. - 275 с.
38. Дейкстра, Э. Структурное программирование / Э. Дейкстра, У. Дал, К. Хоор. – М.: Мир, 1975. – 247 с.
39. Ершов, А.П. Предварительные соображения о лексиконе программирования / А.П. Ершов // Кибернетика и вычислительная техника. - 1984. - С. 199-210.
40. Зайцев, Н.Г. Технология обработки данных в языковой форме / Н.Г. Зайцев. - Киев: Техника, 1989. - 183 с.
41. Зенков, А. И. Разработка подхода к созданию специализированных систем визуализации для высокопроизводительных научных вычислений / А. И. Зенков // Вопросы атомной науки и техники. Математическое моделирование физических процессов. - 2003 - N 4. - С. 81-86.

42. Зюбин, В.Е. Графические и текстовые формы спецификации сложных управляющих алгоритмов: непримиримая оппозиция или кооперация? [Электронный ресурс] / В.Е. Зюбин // Режим доступа: http://www.softcraft.ru/design/gtfs/index.shtml
43. Зюбин, В.Е. Исследование условий применимости языка параллельного программирования СПАРМ для задач построения надежных управляющих программ [Электронный ресурс] / В.Е. Зюбин // Режим доступа: http://www.softcraft.ru/theory/safety/index.shtml
44. Калинов, А.Я. Команда «шаг» в параллельных отладчиках / А.Я. Калинов, К.А. Карганов, К.В. Хоренко // Труды Института системного программирования РАН. - 2004. - N 5 - С. 89-101.
45. Карпов, Ю.Г. Анализ корректности параллельной программы разделения множеств / Ю.Г. Карпов // Программирование. - 1996. - N 6. - С. 27-33.
46. Карпов, Ю.Г. MODEL CHECKING. Верификация параллельных и распределенных программных систем / Ю.Г. Карпов. - СПб.: БХВ-Петербург, 2010. - 560 с.
47. Кларк, Э.М. Верификация моделей программ / Э.М. Кларк, О. Грамберг, Д. Пелед. - М.: МЦНМО, 2002. - 416 с.
48. Клепиков, В.И. Подчиненные сети Петри в задачах логического управления / В.И. Клепиков // Авиакосмическое приборостроение. - 2007. - N 8. - С. 36-40.
49. Ключников, И.Г. Выявление и доказательство свойств функциональных программ методами суперкомпиляции [Электронный ресурс] / И.Г. Ключников // Режим доступа: http://www.keldysh.ru/council/1/klyuchnikov.pdf
50. Ключников, И.Г. Суперкомпиляция функций высших порядков [Электронный ресурс] / И.Г. Ключников // Режим доступа: http://psta.psiras.ru/read/psta2010_3_37-71.pdf
51. Коваленко, М.Р. Интерактивная отладка MPI-программ с использованием параллельного отладчика TDB [Электронный ресурс] / М.Р. Коваленко //

Режим доступа: http://skif.pereslavl.ru/skif/index.cgi?module=chap&action=getpage&data=publications\pub2005\tdb_abrau_2005\tdb_abrau_2005.html

52. Коновалов, Н. Параллельные программы для вычислительных кластеров и сетей [Электронный ресурс] / Н. Коновалов, В. Крюков // Режим доступа: http://www.osp.ru/os/2002/03/181251
53. Костюхин, К.А. Отладка систем реального времени. Обзор [Электронный ресурс] / К.А. Костюхин // Режим доступа: http://www.citforum.ru/programming/digest/rtsdebug.shtml
54. Крюков, В.А. Автоматизация отладки параллельных программ / В.А. Крюков, М.В. Кудрявцев // Вычислительные методы и программирование. - 2006. - N 7. - С. 102-110.
55. Крюков, В.А. Разработка параллельных программ для вычислительных кластеров и сетей. Проблемы и перспективы [Электронный ресурс] / В.А. Крюков // Режим доступа: http://www.kiam.ru/dvm/dvmhtm1107/publishr/clust-141201.htm
56. Кузьмин, Е.В. О верификации автоматных программ / Е.В. Кузьмин, В.А. Соколов // Актуальные проблемы информатики и математики. Сборник статей к 20-летию факультета им. П.Г. Демидова. - 2006. - С. 27-32.
57. Кузьмин, Е.В. О дисциплине специализации «Верификация Программ» / Е.В. Кузьмин, В.А. Соколов // Преподавание математики и компьютерных наук в классическом университете. Доклады II-й научно-метод. конференции преподавателей матем. ф-та и ф-та ИВТ Ярославского гос. ун-та им. П.Г. Демидова. - 2007. - С. 91-101.
58. Кулямин, В.В. Методы верификации программного обеспечения [Электронный ресурс] / В.В. Кулямин // Режим доступа: http://panda.ispras.ru/~kuliamin/docs/VerMethods-2008-ru.pdf
59. Кулямин, В. Формальная верификация: методы и приложения [Электронный ресурс] / В. Кулямин, Е. Корныхин // Режим доступа: http://panda.ispras.ru/~kuliamin/docs/FV-2006-talk-ru.ppt

60. Ластовецкий, А.Л. Анализ структурной эквивалентности описаний в компиляторе с языка С. / А.Л. Ластовецкий, И.Н.Ледовский // Вопросы кибернетики: Приложения системного программирования, Научный совет по комплексной проблеме "Кибернетика" РАН. - 1995. - С. 6-19.
61. Ластовецкий, А.Л. Расширение ANSI C для векторных и суперскалярных компьютеров. / А.Л. Ластовецкий, И.Н.Ледовский, С.С.Гайсарян, Д.А.Халецкий // Программирование. - 1995. - N 1. - С. 26-36.
62. Левченко, В. Д. Асинхронные параллельные алгоритмы как способ достижения 100% эффективности вычислений [Электронный ресурс] / В. Д. Левченко // Режим доступа: http://www.softcraft.ru/parallel/asyncell/index.shtml
63. Легалов, А.И. Инструментальная поддержка процесса разработки эволюционно расширяемых параллельных программ / А.И. Легалов // Проблемы информатизации региона. ПИР-2003. Материалы 8-й Всероссийской научно-практической конференции. - Красноярск, 2003. - С. 132-136.
64. Легалов, А.И. Использование асинхронных вычислений в функциональных языках параллельного программирования / А.И. Легалов // Распределенные и кластерные вычисления. Избранные материалы четвертой школы-семинара. Институт вычислительного моделирования СО РАН. - Красноярск, 2005. - С. 172-183.
65. Легалов, А.И. Модель параллельных вычислений функционального языка / А.И. Легалов, Ф.А. Казаков, Д.А. Кузьмин, А.И. Водяхо // Известия ГЭТУ. Структуры и математическое обеспечение специализированных средств. - С.-Петербург, 1996. - Вып. 500. - С. 56-63.
66. Легалов, А. И. Модель функционально-потоковых вычислений и язык программирования «Пифагор» / А.И. Легалов, Ф.А. Казаков, Д.А. Кузьмин, Д. В. Привалихин // Вторая школа семинар. Распределенные и кластерные вычисления. Избранные материалы. - Красноярск, 2002. - С. 101-120.

67. Легалов, А.И. На пути к переносимым параллельным программам. / А.И. Легалов, Ф.А. Казаков, Д.А. Кузьмин, Д. В. Привалихин // Открытые системы. - 2003. - N 5. - С. 36-42.
68. Легалов, А.И. Особенности функционального языка параллельного программирования «Пифагор» / А.И. Легалов, Д.В. Привалихин // Высокопроизводительные вычисления на кластерных системах. Материалы четвертого Международного научно-практического семинара и Всероссийской молодежной школы. - Самара, 2004. - С. 173-179.
69. Легалов, А.И. Параллельное программирование в языках Haskell и Пифагор / А.И. Легалов, Ф.А. Казаков // Проблемы информатизации региона. ПИР-2001: Сб. науч. Трудов. - Красноярск: ИПЦ КГТУ, 2002. - С. 48-55.
70. Легалов, А.И. Параллельный язык управления потоков данных / А.И. Легалов, Ф.А. Казаков, Д.А. Кузьмин // Математическое обеспечение и архитектура ЭВМ: Сб. научных работ. - Красноярск, 1997. - N 2. - С. 105-113.
71. Легалов, А.И. Потоковая модель параллельных вычислений / А.И. Легалов, Ф.А. Казаков, Д.А. Кузьмин // Вестник Красноярского государственного технического университета. - 1996. - N 6. - С. 60-67.
72. Легалов, А.И. Событийная модель вычислений, поддерживающая выполнение функционально-потоковых параллельных программ / А.И. Легалов, Г.В. Савченко, В.С. Васильев // Системы. Методы. Технологии. - 2012. - N 1(13). - С. 113-119.
73. Легалов, А.И. Стратегии управления в вычислительных системах и языках программирования / А.И. Легалов // Распределенные и кластерные вычисления. Избранные материалы Школы-семинара. Институт вычислительного моделирования СО РАН. - Красноярск, 2001. - С. 94-108.
74. Легалов, А.И. Управление в вычислительных системах и языках программирования / А.И. Легалов // Материалы конференции "Проблемы информатизации города". - Красноярск, 1994. - С. 198-202.

75. Липаев, В.В. Качество программного обеспечения / В.В. Липаев. - М: Финансы и статистика, 1983. - 262 с.

76. Лисков, Б. Использование абстракций и спецификаций при разработке программ / Б. Лисков, Дж. Гатэг. – М.: Мир, 1989. - 424 с.

77. Любченко, В.С. К проблеме создания модели параллельных вычислений [Электронный ресурс] / В.С. Любченко // Режим доступа: http://www.softcraft.ru/auto/ka/modproblem/modproblem.pdf

78. Малышкин, В.А. Основы параллельных вычислений / В.А. Малышкин. - Новосибирск: Наука, 1999. - 72 с.

79. Массер, Д. Спецификация абстрактных типов данных в системе AFFIRM / Д. Массер // Требования и спецификации в разработке программ. - 1984. - С. 199-222.

80. Накадзима, Р. Иерархическая спецификация и верификация программ / Р. Накадзима, М. Хонда, Х. Накахара // Требования и спецификации в разработке программ - 1984. - С. 135-164.

81. Непомнящий, В.А. Верификация программ обработки данных с использованием автоматной модели файлов / В.А. Непомнящий, О.М. Рякин // Прикладная информатика. - 1988. - N 14. - С. 180-190.

82. Непомнящий, В.А. Верификация программ обработки файлов / В.А. Непомнящий // Программирование. - 1981. - N 2 - С. 34-43.

83. Непомнящий, В.А. Верификация программ сортировки массивов / В.А. Непомнящий, Т.Г. Чурина // Языки и системы программирования. - Новосибирск: ВЦ СО АН СССР, 1979. - С. 21-36.

84. Непомнящий, В.А. Доказательство правильности программ линейной алгебры / В.А. Непомнящий // Программирование. - 1982. - N 4. - С. 63-72.

85. Непомнящий, В.А. На пути к верификации С программ. Часть 1. Язык C-light / В.А. Непомнящий, И.С. Ануреев, И.Н. Михайлов, А.В. Промский. - Новосибирск: Известия РАН. Сиб. Отд-ние. ИСИ, 2001. - 49 с.

86. Непомнящий, В.А. На пути к верификации С программ. Часть 2. Язык C-light-kernel и его аксиоматическая семантика / В.А. Непомнящий, И.С.

Ануреев, И.Н. Михайлов, А.В. Промский. - Новосибирск: Известия РАН. Сиб. Отд-ние. ИСИ, 2001. - 58 с.

87. Непомнящий, В.А. Об одном подходе к спецификации и верификации транслятора / В.А. Непомнящий, А.А. Сулимов // Программирование. - 1982. - N 4. - С. 51-58.
88. Непомнящий, В.А. Практические методы верификации программ / В.А. Непомнящий // Кибернетика. - 1984. - N 2. - С. 21-28.
89. Непомнящий, В.А. Прикладные методы верификации программ / В.А. Непомнящий, О.М. Рякин. - М: Радио и связь, 1988. - 255 с.
90. Непомнящий, В.А. Проблемно-ориентированная верификация программ / В.А. Непомнящий // Программирование. - 1986. - N 1. - С. 3-13.
91. Непомнящий, В.А. Проблемно-ориентированные базы знаний и их применение в системе верификации программ СПЕКТР / В.А. Непомнящий, А.А. Сулимов // Известия РАН. Сер. "Теория и системы управления". - 1997. - N 2. - С. 169-175.
92. Непомнящий, В.А. Элиминация инвариантов циклов при верификации программ / В.А. Непомнящий // Программирование. - 1985. - N 3. - С. 3-13.
93. Орехов, А. И. Логическое программирование в Mozart [Электронный ресурс] / А. И. Орехов // Режим доступа: http://www.softcraft.ru/paradigm/logmozart/index.shtml
94. Орехов, А. И. Приемы параллельного программирования в Mozart 1.3.1 [Электронный ресурс] / А. И. Орехов // Режим доступа: http://www.softcraft.ru/parallel/parmozart/index.shtml
95. Отладка программ в OpenMP [Электронный ресурс] Режим доступа: http://www.intuit.ru/department/se/openmp/5/3.html
96. Отладка MPI программ в TotalView [Электронный ресурс] – Режим доступа:http://www.math.spbu.ru/user/rus/cluster/Doc/Library/mp_user/mp_user7.shtml

97. Отладчик buddha - A declarative debugger for Haskell 98 [Электронный ресурс] Режим доступа: http://www.berniepope.id.au/docs/BerniePope.PhD.Thesis.pdf
98. Отладчик GDB [Электронный ресурс] Режим доступа: http://www.gnu.org/software/gdb
99. Отладчик Hat – The Haskell Tracer [Электронный ресурс] Режим доступа: http://code.haskell.org/hat/docs/
100. Отладчик HOOD – The Haskell Object Observation Debugger [Электронный ресурс] Режим доступа: http://www.berniepope.id.au/docs/BerniePope.PhD.Thesis.pdf
101. Отладчик Ozcar [Электронный ресурс] Режим доступа: http://doc.rz.ifi.lmu.de/programming/mozart/mozart/doc/ozcar/index.html
102. Отладчик Panorama [Электронный ресурс] Режим доступа: http://www.cs.ucsd.edu/users/berman/panorama
103. Отладчик TotalView [Электронный ресурс] Режим доступа: http://www.roguewave.com/products/totalview.aspx
104. Паршин, П.В. Программный инструментарий для автоматизированной верификации и анализа программного обеспечения / П.В. Паршин, И.А. Лягин, А.В. Николаев, Г.Н. Чижухин // Проблемы информационной безопасности. Компьютерные системы. - 1999. - N 1. - С. 53-56.
105. Петренко, А.К. Методы отладки и мониторинга параллельных программ / А.К. Петренко // Программирование. - 1994. - N 3. - С. 39-63.
106. Редькин, А.В. Промежуточное представление информационного графа для языка Пифагор / А.В. Редькин // Информатика и информационные технологии: Материалы межвузовской научной конференции. − Красноярск: ИПЦ КГТУ, 2004. - С. 52-60.
107. Редькин, А.В. Промежуточное представление языка потокового программирования / А.В. Редькин // Сибирская школа-семинар по параллельным вычислениям - Томск: изд-во Том. ун-та, 2006. - С. 121-124.

108. Редькин, А.В. Событийный процессор для функционально-потоковых параллельных вычислений / А.В. Редькин, А.В. Матковский // Материалы XI Всероссийской научно-практической конференции Проблемы Информатизации Региона (ПИР-2009) - Красноярск, 2009. - С. 151-153.

109. Рекурсивный функциональный язык программирования РЕФАЛ [Электронный ресурс] - Режим доступа: www.refal.net

110. Робачевский, А.М. Операционная система Unix: учеб. пособие для вузов / А.М. Робачевский. – СПб.: БХВ–Петербург, 1999. – 528 с.

111. Руководство по программированию на MPI для МВС-1000М [Электронный ресурс] Режим доступа: http://www2.sscc.ru/MVS-1000/MVS-128/MPrg_Guide/1-3_2.htm

112. Рякин, О.М. Основы методологии проектирования корректных программ / О.М. Рякин. - М.: МЭИ, 1980. - 82 с.

113. Синицын, С.В. Верификация программного обеспечения / С.В. Синицын, Н.Ю. Налютин. - М.:БИНОМ, 2008. - 368 с.

114. Система программирования Mozart и язык Oz [Электронный ресурс] Режим доступа: http://www.mozart-oz.org

115. Смелянский, Р.Л. Модель функционирования распределенных вычислительных систем. / Р.Л. Смелянский // Вестн. Моск. Ун-та. Вычислительная математика и Кибернетика. - 1990. - N 3. - С. 3-21.

116. Смелянский, Р.Л. Об инварианте поведения программ / Р.Л. Смелянский // Вестн. Моск. Ун-та. Вычислительная математика и Кибернетика. - 1990. - N 4. - С. 54-60.

117. Теренс, Чан. Системное программирование на C++ для UNIX / Чан Теренс. – К.: Издательская группа BHV, 1999. – 592 с.

118. Тыугу, Э.Х. Концептуальное программирование / Э.Х. Тыугу. – М.: Наука, 1984. - 256 с.

119. Удалова, Ю.В. Верификация и спецификация программ на функционально-потоковом параллельном языке Пифагор / Ю.В. Удалова //

Международная Ершовская конференция по информатике, рабочий семинар «Наукоемкое программное обеспечение». - Новосибирск, 2011. - С. 259-264.

120. Удалова, Ю.В. Методы отладки и верификации функционально-потоковых параллельных программ / Ю.В. Удалова, А.И. Легалов, Н.Ю. Сиротинина // Журнал Сибирского федерального университета. Техника и технологии. - 2011. - N 2. - С. 213-224.

121. Удалова, Ю.В. Отладка программ на функционально-потоковом параллельном языке Пифагор с подстановкой интервальных значений / Ю.В. Удалова, А.И. Легалов, Н.Ю. Сиротинина // Алтайский государственный технический университет им. И.И. Ползунова, Ползуновский вестник. - 2013. - N 2. - С. 46-48.

122. Функциональный язык Haskell [Электронный ресурс] Режим доступа: http://www.haskell.org

123. Функциональный язык SISAL [Электронный ресурс] Режим доступа: http://sisal.sourceforge.net/

124. Хан, Ю. Обзор средств отладки программ на функциональных языках [Электронный ресурс] / Ю. Хан // Режим доступа: http://db.iis.nsk.su/preprints/articles/pdf/sbor_kas_12_han.pdf

125. Хоар, Ч. Взаимодействующие последовательные процессы / Ч. Хоар. - М.: Мир, 1989. - 264 с.

126. Царьков, Д.В. Использование модульности при верификации распределенных программ / Д.В. Царьков // "Программные системы и инструменты": Тематический сборник факультета ВМиК МГУ им. Ломоносова. - М.: МАКС Пресс, 2000. - N 1. - С. 128-136.

127. Царьков, Д.В. Система формальной верификации распределенных программ в среде имитационного моделирования DYANA / Д.В. Царьков // Труды Международной конференции "Параллельные вычисления и задачи управления" (PACO'2001). - М.: Институт проблем управления им. В.А.Трапезникова РАН, 2001. - С.161-182.

128. Цилюрик, О. QNX/UNIX: анатомия параллелизма / О. Цилюрик, Е. Горошко. - СПб.: Символ-Плюс, 2006. - 288 с.

129. Чень. Математическая логика и автоматическое доказательство теорем / Чень. – М.: Наука, 1983. - 358 с.

130. Шошмина, И.В. Введение в язык Promela и систему комплексной верификации Spin / И.В. Шошмина, Ю.Г. Карпов. - Санкт-Петербургский государственный политехнический университет, 2009. - 66 с.

131. Язык спецификации HOL [Электронный ресурс] Режим доступа: http://www.cl.cam.ac.uk/research/hvg/HOL

132. Baier, C. Principles of Model Checking [Электронный ресурс] / C. Baier, J. Katoen // Режим доступа: http://is.ifmo.ru/books/_principles_of_model_checking.pdf

133. Ennals, R. HsDebug: Debugging Lazy Programs by Not Being Lazy [Электронный ресурс] / R. Ennals, S. Jones // Режим доступа: http://berkeley.intel-research.net/~rennals/pubs/hw2003.pdf

134. May, J. Designing a Parallel Debugger for Portability [Электронный ресурс] / J. May // Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.3713&rep=rep1&type=pdf

Приложение А

Примеры отладки функционально-потоковых параллельных программ в инструментальной среде

А.1 Отладка в пошаговом режиме

Отладка производится над функцией вычисления модуля числа Abs.

Abs << funcdef Param

{ ({Param:-}, Param):[(Param,0): (<,>=):?] >> return ; }

Выбран пошаговый режим обхода с отображением графа (рисунок А.1), поддержкой задержанных вычислений и текстовым отображением отладки в режиме «операторы».

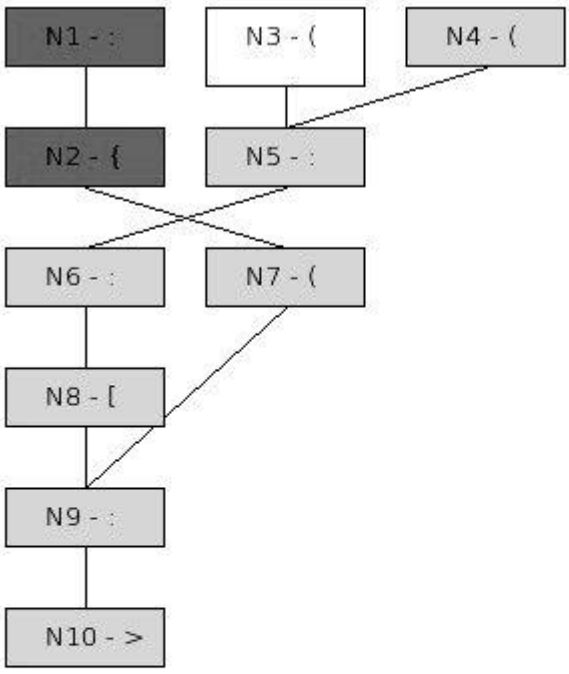


Рисунок А.1 – Граф функции Abs

На графе (рисунок А.1) вершины-операторы №1 и №2 являются задержанными и выделяются специальным цветом, вершина №3 уже вычислена отладчиком, и при нажатии на нее можно увидеть и сам оператор и его значение. Запись оператора в программе отображается не только при активации вершины графа, но и внизу изображения графа (рисунок А.1), автоматически генерируемого инструментальной средой.

Вершина-оператор №3 оказалась вычисленной на первом шаге отладки (рисунок А.2) из-за включения поддержки задержанных вычислений.

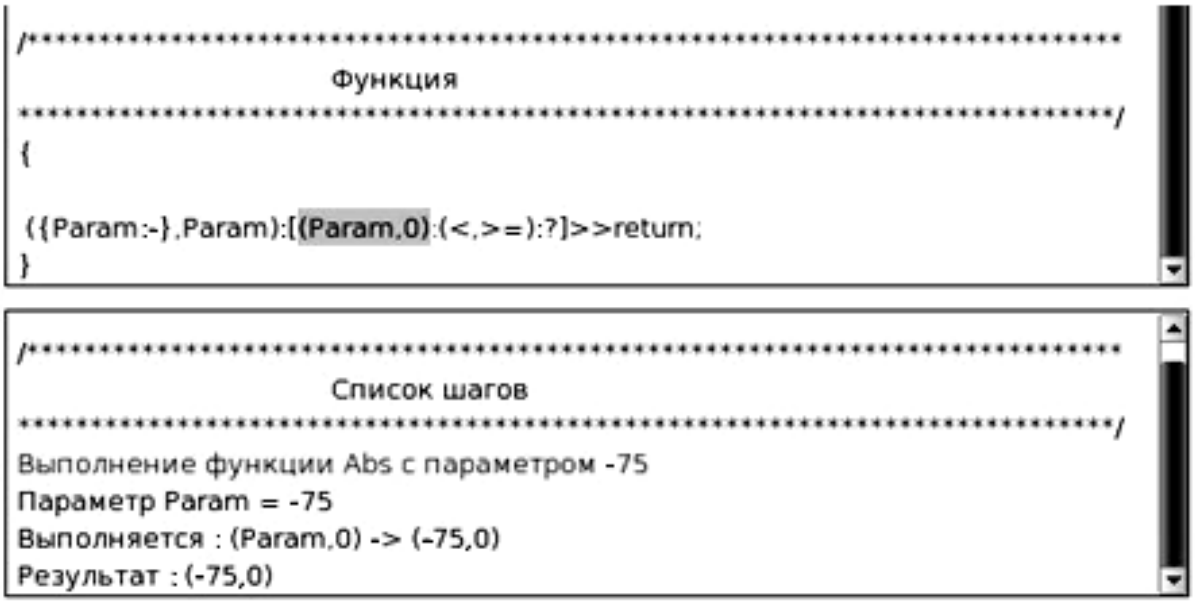


Рисунок А.2 – Текстовое представление первого шага отладки функции Abs

Текстовое отображение отладки выделяет в тексте программы рассматриваемую функцию, помещая перед ней текстовый заголовок, также в коде самой функции оператор, выбранный для вычисления на шаге отладки, выделяется цветом (рисунок А.2).

Последующие шаги отладки будут отображены в графическом и текстовом режимах следующим образом (рисунки А.3 – А.13).

На рисунке А3 следующий шаг отладки перейдет от оператора (Param, 0) к оператору (<, >=).

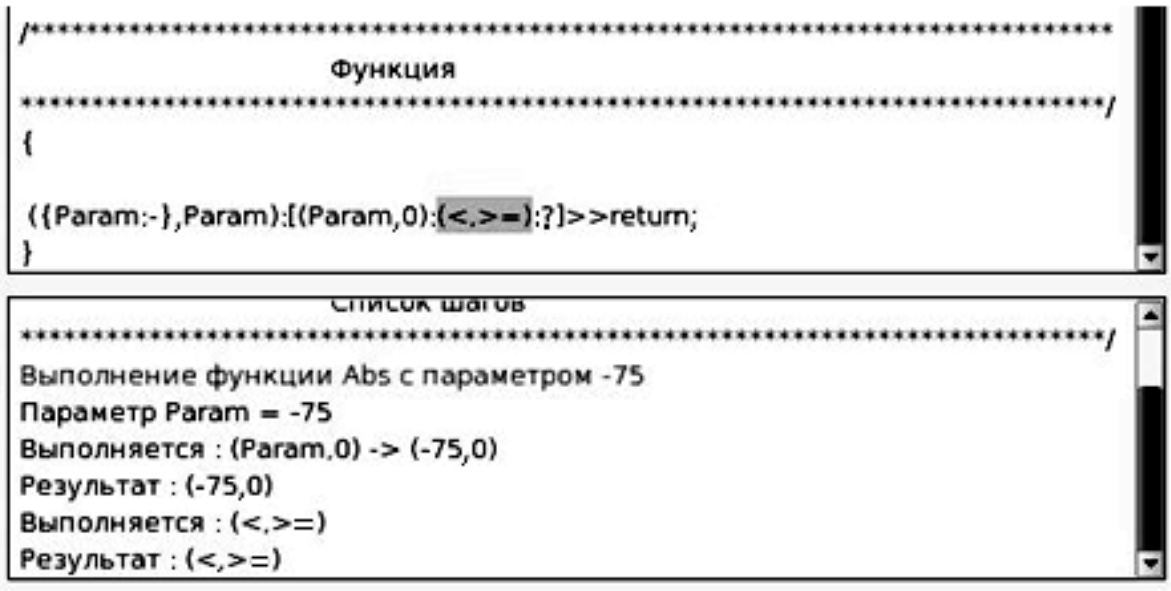


Рисунок А.3 – Текстовое представление второго шага отладки функции Abs

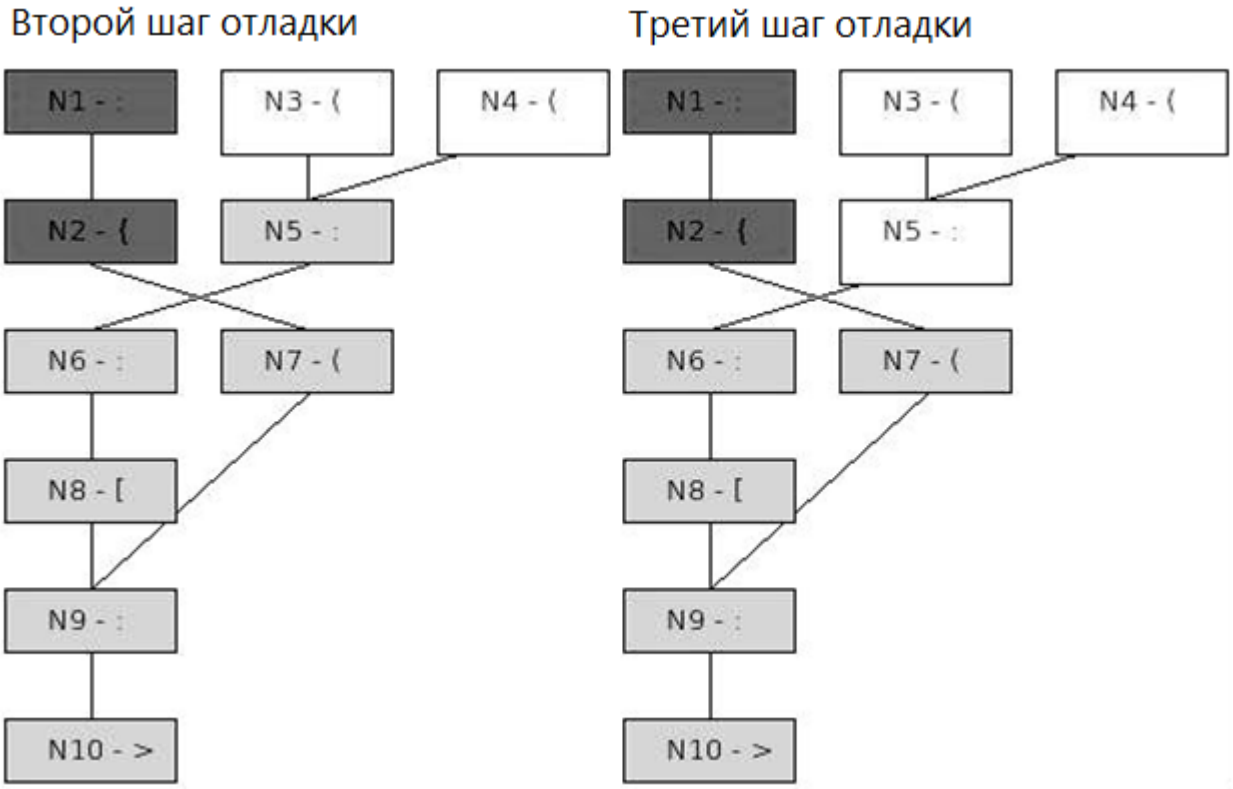


Рисунок А.4 – Графическое представление второго и третьего шагов отладки функции Abs

На втором шаге отладки (рисунок А.4) вычислены операторы №3 и №4, на третьем шаге отладки (рисунок А.4) выполнилось вычисление оператора № 5.

На третьем шаге отладки (рисунок А.5) приходит очередь вычисления оператора интерпретации (Param,0):(<,>=).

```
/**************************************************************************
                        Функция
**************************************************************************/
{

({Param:-},Param):[(Param,0):(<,>=):?]>>return;
}
```

```
Выполнение функции Abs с параметром -75
Параметр Param = -75
Выполняется : (Param,0) -> (-75,0)
Результат : (-75,0)
Выполняется : (<,>=)
Результат : (<,>=)
Выполняется : (Param,0):(<,>=) -> (-75,0):(<,>=)
Результат : (true,false)
```

Рисунок А.5 – Текстовое представление третьего шага отладки функции Abs

На пятом шаге отладке (рисунок А.6) вычисляется оператор №7, открывающий при отладке задержанные операторы №1 и №2.

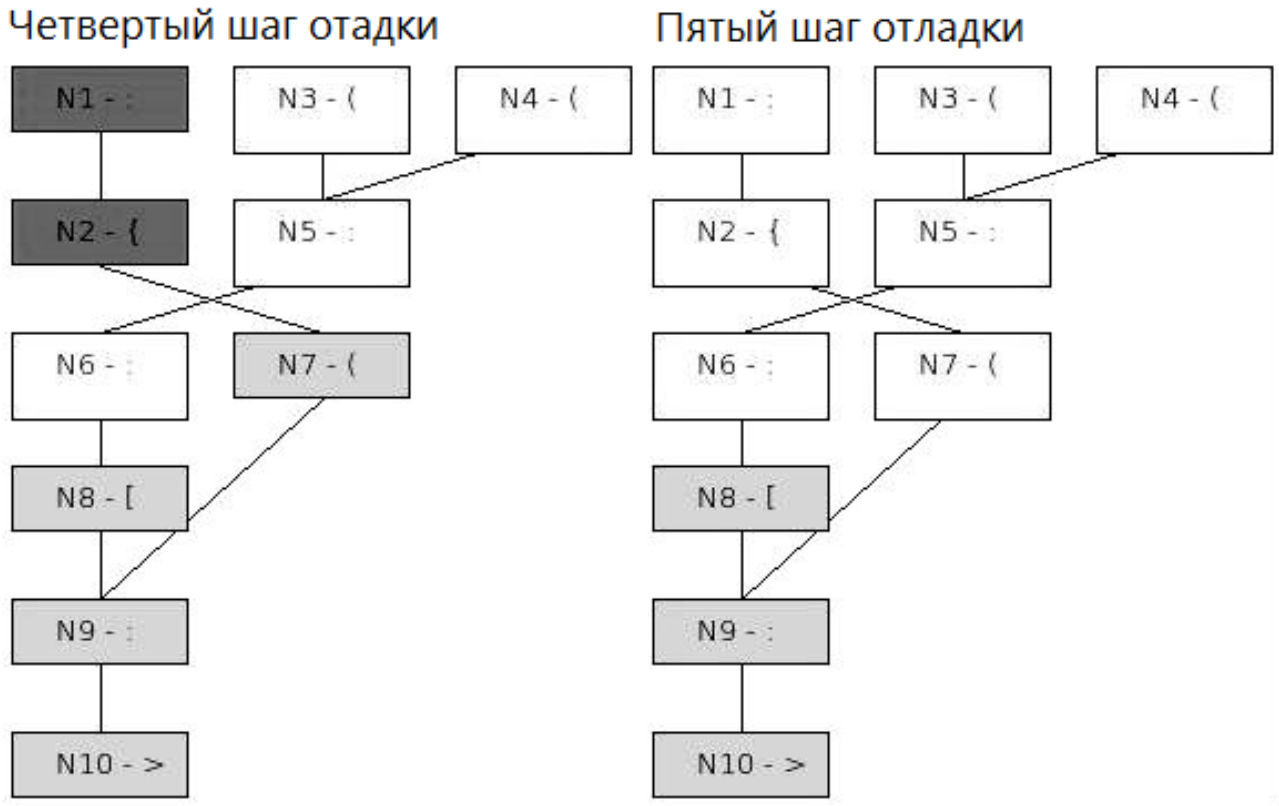


Рисунок А.6 – Графическое представление четвертого и пятого шагов отладки

На четвертом шаге отладки (рисунок А.7) приходит очередь вычисления оператора (Param,0):(<,>=):?, что после всех подстановок даст список = [1].

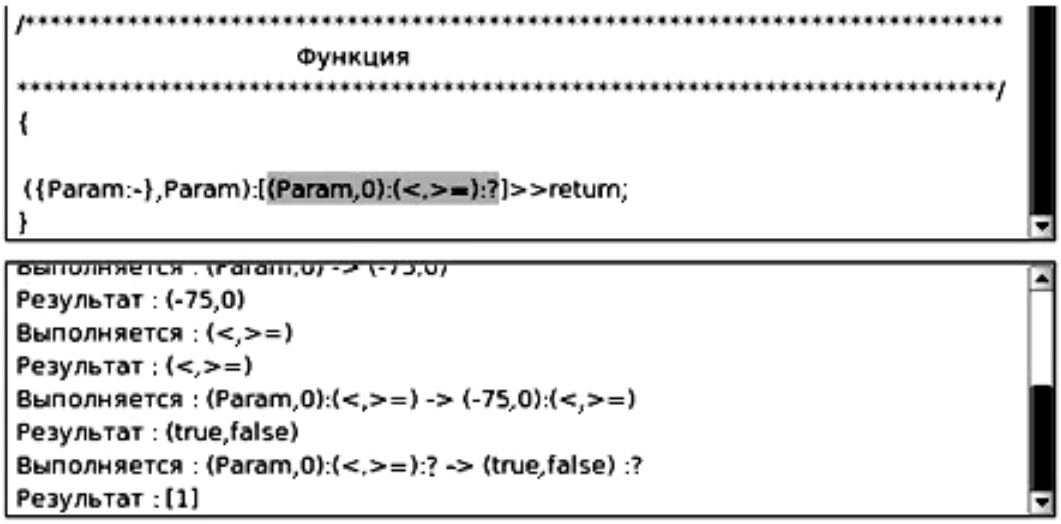


Рисунок А.7 – Текстовое представление четвертого шага отладки функции Abs

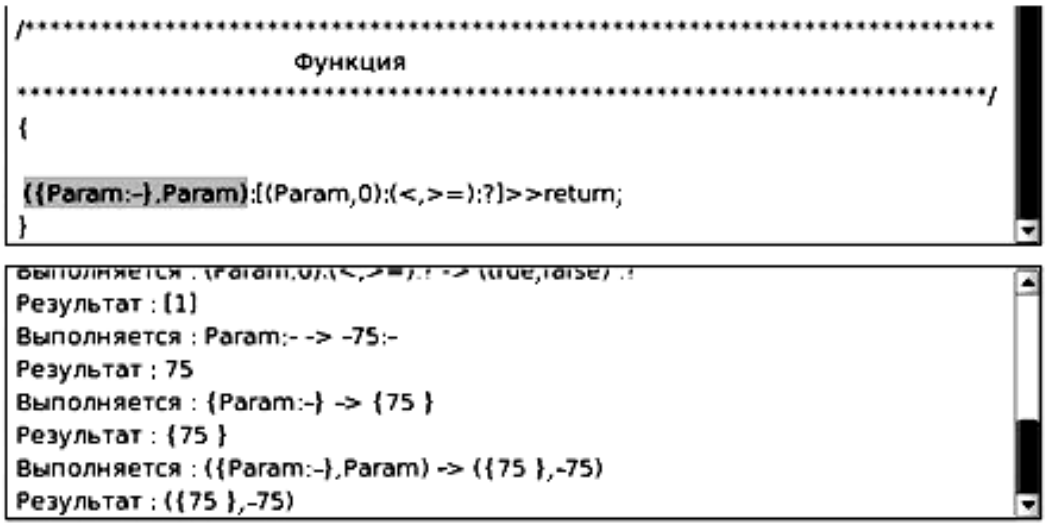


Рисунок А.8 – Текстовое представление пятого шага отладки функции Abs

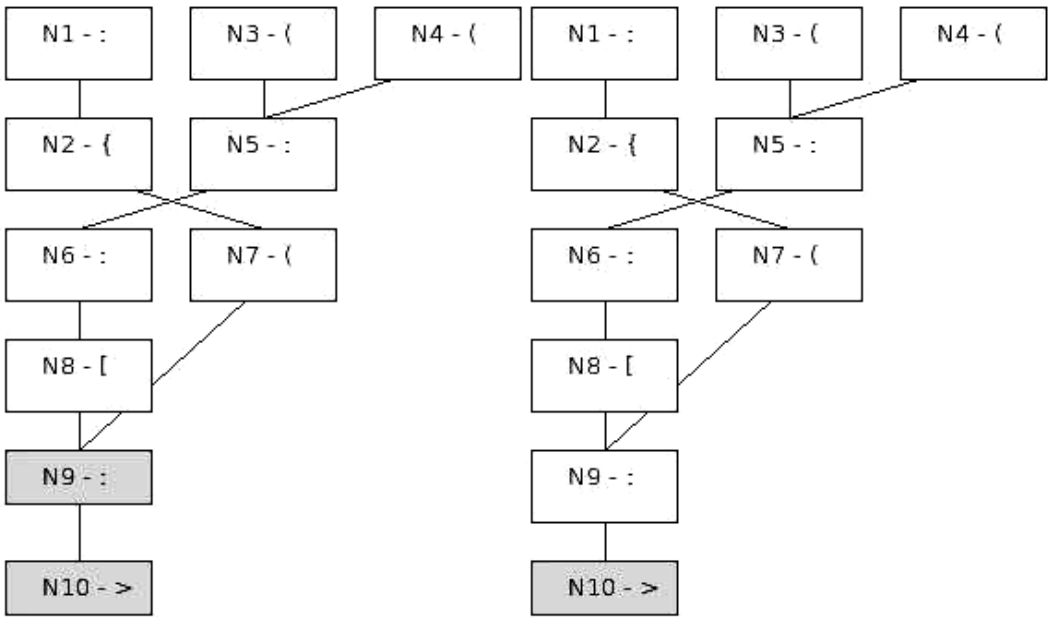


Рисунок А.9 – Графическое представление шестого и седьмого шагов отладки

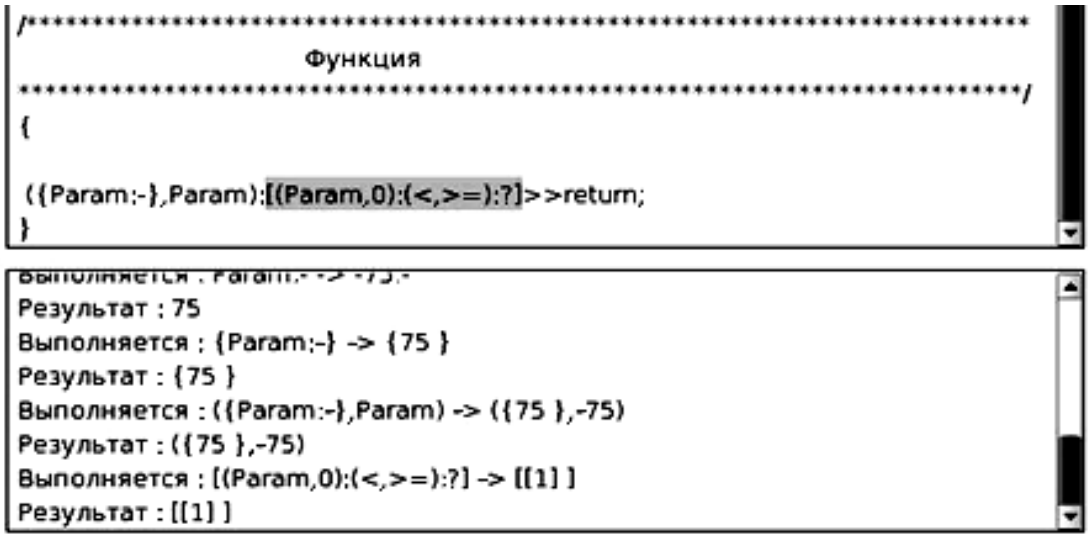


Рисунок А.10 – Текстовое представление шестого шага отладки функции Abs

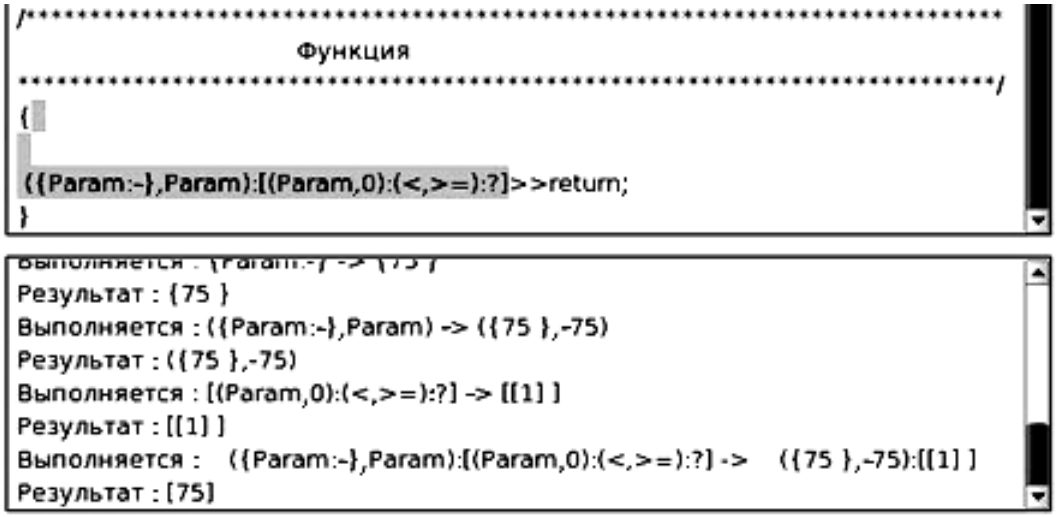


Рисунок А.11 – Текстовое представление седьмого шага отладки функции Abs

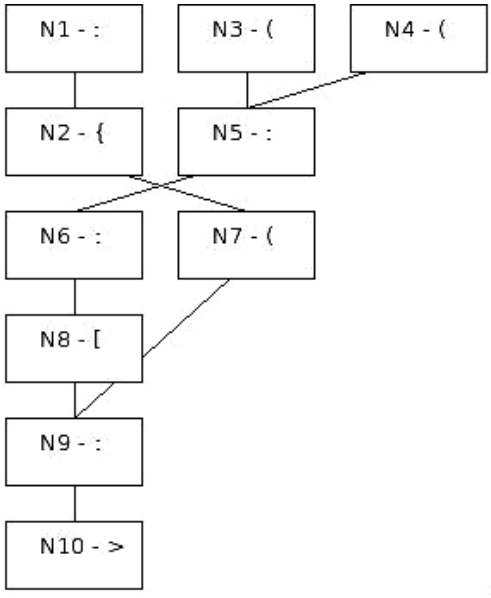


Рисунок А.12 – Графическое представление заключительного шага отладки

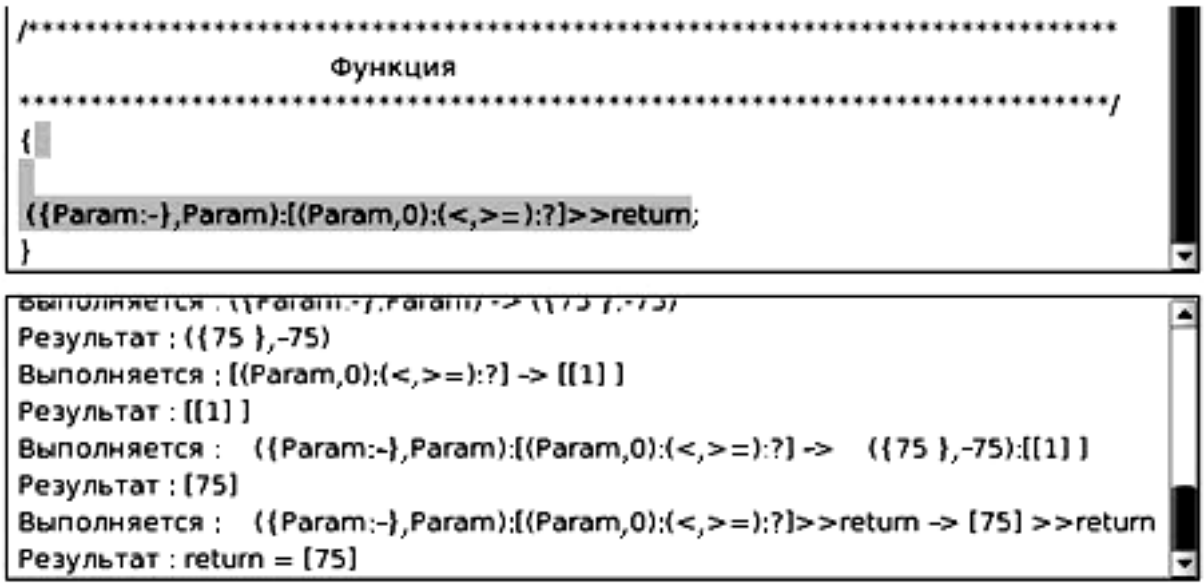


Рисунок А.13 – Текстовое представление заключительного шага

Пошаговая отладка функции Abs с указанием пользователя вычислять задержанные операторы как обычные прошла бы за десять шагов и началась с вычисления оператора №1.

А.2 Отладка ветвей

Отладка производится над функцией cn, получающей число и возвращающей двойку, если число четное, а в противном случае возвращающей единицу.

```
cn << funcdef x
{ ( (x,2) : % : 2, 0) : (!=,=) : ? >> return ; }
```

Выбран режим отладки ветвей с отображением графа (рисунок А.14) и текстовым представлением отладки в режиме «свертка функций». Такой режим отображает на экране код отлаживаемой функции с выделенными операторами, обрабатываемыми на текущем шаге отладки, и код функции с подставленными вычисленными значениями операторов на текущем и предыдущих шагах отладки. На каждом новом шаге предыдущее отображение текста программы стирается. Выделение операторов в тексте осуществляется с помощью изменения цвета шрифта, для лучшей видимости на нижестоящих рисунках выделение операторов отображается подчеркиванием.

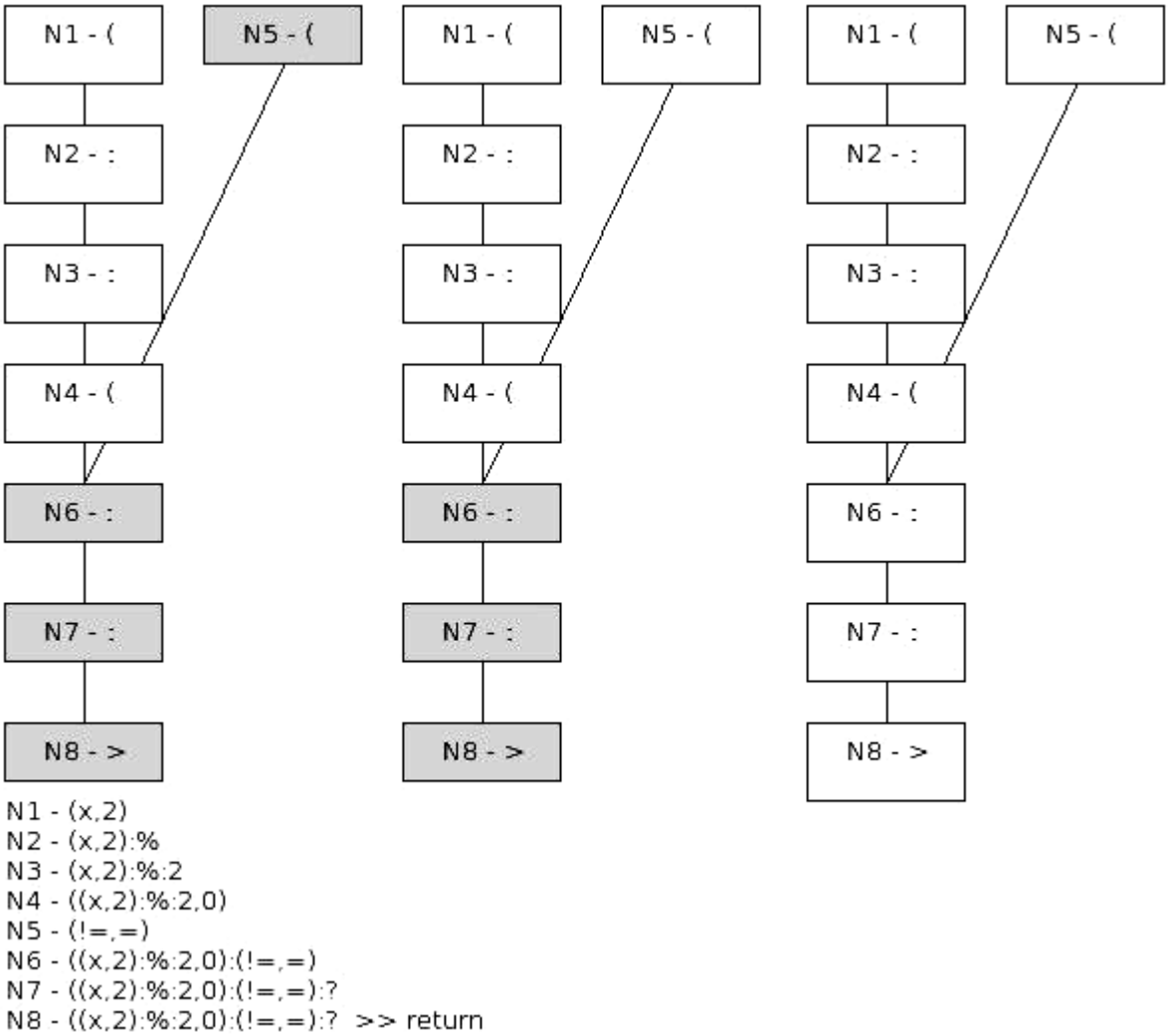


Рисунок А.14 – Графы функции cn на трех шагах отладки

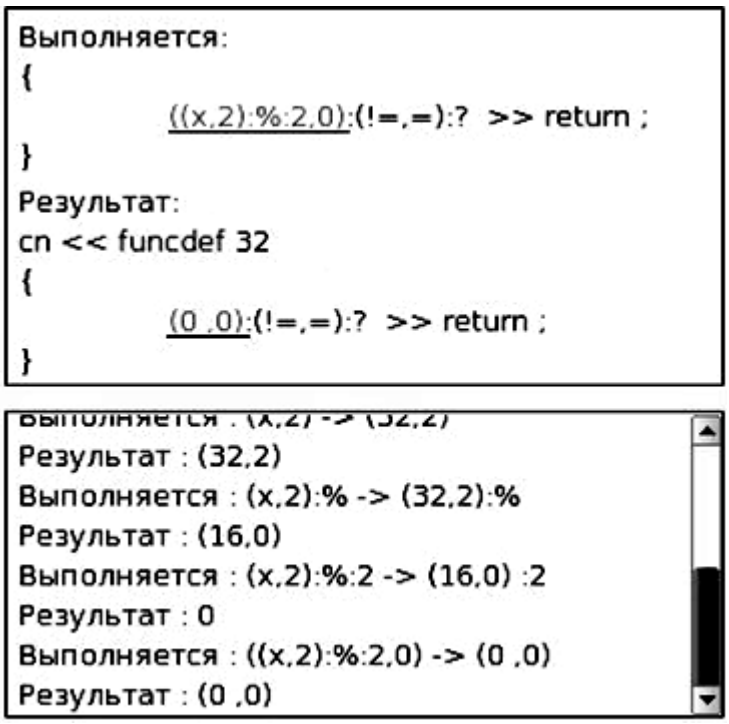


Рисунок А.15 – Текстовое представление первого шага отладки функции cn

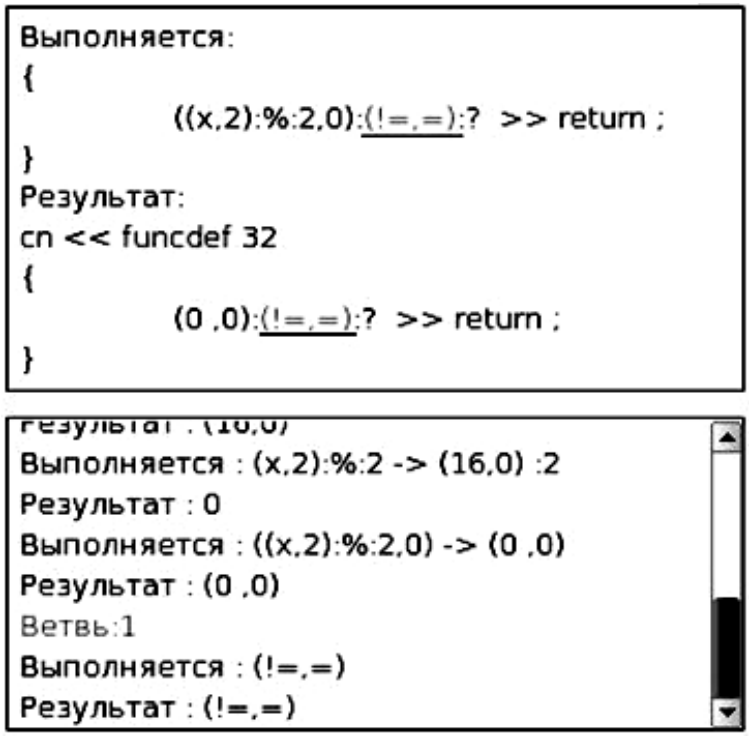


Рисунок А.16 – Текстовое представление второго шага отладки функции cn

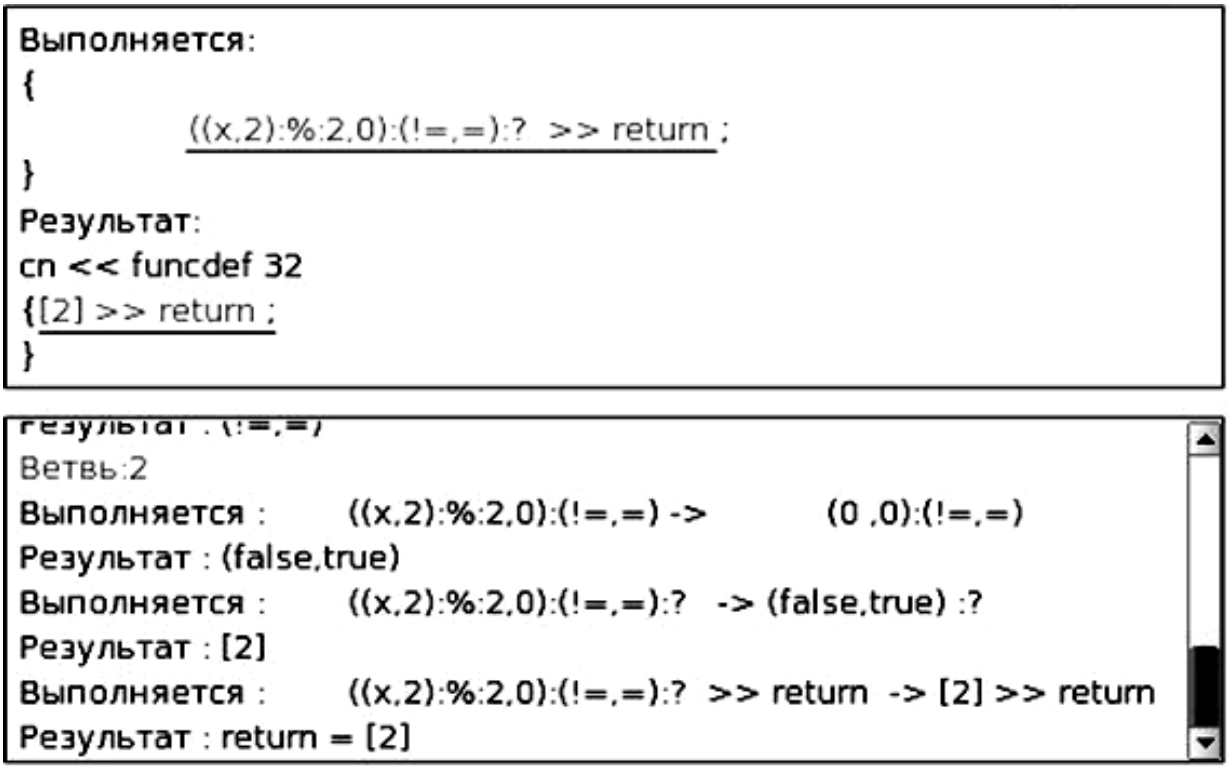


Рисунок А.17 – Текстовое представление последнего шага отладки функции cn

Первый шаг отладки (рисунок А.15) выполняет ветвь операторов №1 - №4 (рисунок А.14), образующих набор команд ((x,2) : % : 2, 0).

Второй шаг отладки (рисунок А.16) выполняет ветвь, состоящую из одного оператора № 5 (рисунок А.14). Это объединение базовых функций в список (!=,

=), чье вычисленное значение аналогично записи оператора в программе и равно (!=, =).

Третий и заключительный шаг отладки (рисунок А.17) вычисляет ветвь графа, состоящую из операторов № 6 – 8 (рисунок А.14), образующих набор команд ((x,2) : % : 2, 0) : (!=,=) : ? >> return, то есть (результат выполнения первого шага отладки) : (результат выполнения второго шага отладки) : ? >> return.

Отладка рассматриваемой функции cn в режиме отладки ветвей завершилась за три шага, в пошаговом режиме потребовалось бы восемь шагов. Однако существуют такие функции, для которых все ветви графа состоят из малого числа операторов и в таких случаях режим отладки ветвей близок по числу шагов отладки к режиму пошаговой отладки.

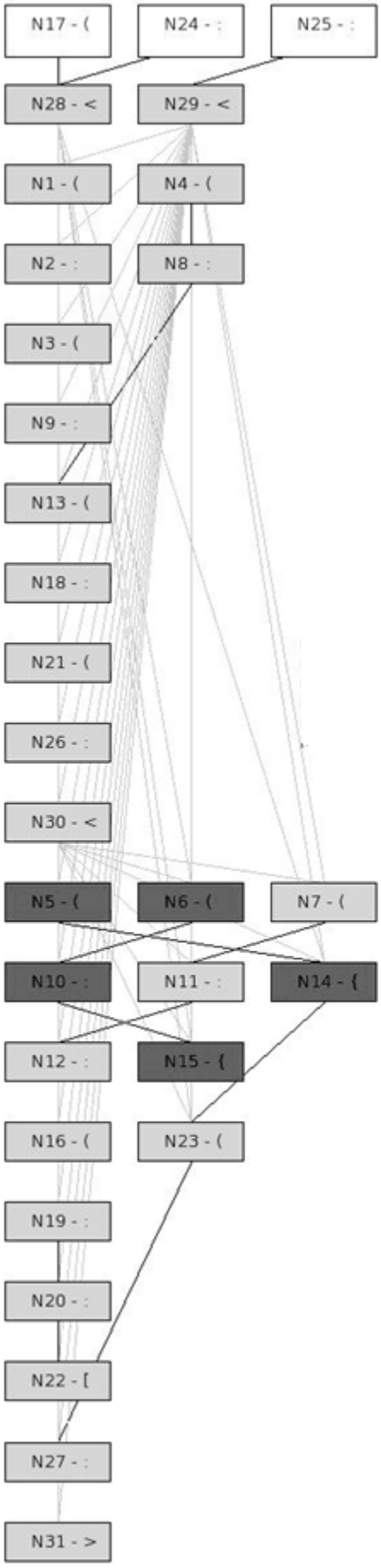
А.3 Отладка слоев

Производится отладка функции поиска корня числа SqrtR.

```
SqrtR << funcdef Param
{
        A << Param:1;
        X << Param:2;
        Xn << (X,(((X,X):*,A):-,(2,X):*):/):-;
        ({(A,Xn)},{(A,Xn):SqrtR}):[((Xn,X):-:Abs,0.00001)
        :(<,>=):?]>>return;
};
```

Выбран режим отладки слоев с отображением графа (рисунок А.18), поддержкой задержанных вычислений и текстовым представлением отладки в режиме «список функций». Такой режим подобен режиму «свертка функции», но отличается от него тем, что сохраняет историю всех вычислений и подстановок в программный код.

На рисунке А.18 операторы графа №5, №6, №10, №14, №15 являются задержанными, а операторы №17, №24, №25 выполнились как образующие первый слой.



N1 - (X, X)
N2 - (X, X):*
N3 - ((X, X):*, A)
N4 - (2, X)
N5 - (A, Xn)
N6 - (A, Xn)
N7 - (Xn, X)
N8 - (2, X):*
N9 - ((X, X):*, A):-
N10 - (A, Xn): SqrtR
N11 - (Xn, X):-
N12 - (Xn, X):-:Abs
N13 - (((X, X):*, A):-, (2, X):*)
N14 - {(A, Xn)}
N15 - {(A, Xn): SqrtR}
N16 - ((Xn,X):-:Abs,0.00001)
N17 - (<, >=)
N18 - (((X, X):*, A):-, (2, X):*):/
N19 - ((Xn,X):-:Abs,0.00001):(<, >=)
N20 - ((Xn,X):-:Abs,0.00001):(<, >=):?
N21 - (X, (((X, X):*, A):-, (2, X):*):/)
N22 - [((Xn,X):-:Abs,0.00001):(<, >=):?]
N23 - ({(A,Xn)},{(A,Xn):SqrtR})
N24 - Param:1
N25 - Param:2
N26 - (X, (((X, X):*, A):-, (2, X):*):/):-
N27 - ({(A,Xn)},{(A,Xn):SqrtR}):[((Xn,X):-:Abs,0.00001):(<,>=):?]
N28 - A << Param:1
N29 - X << Param:2
N30 - Xn << (X,(((X,X):*,A):-,(2,X):*):/):-
N31 - ({(A,Xn)},{(A,Xn):SqrtR}):[((Xn,X):-:Abs,0.00001):(<,>=):?]>>return

Рисунок А.18 – Граф функции SqrtR

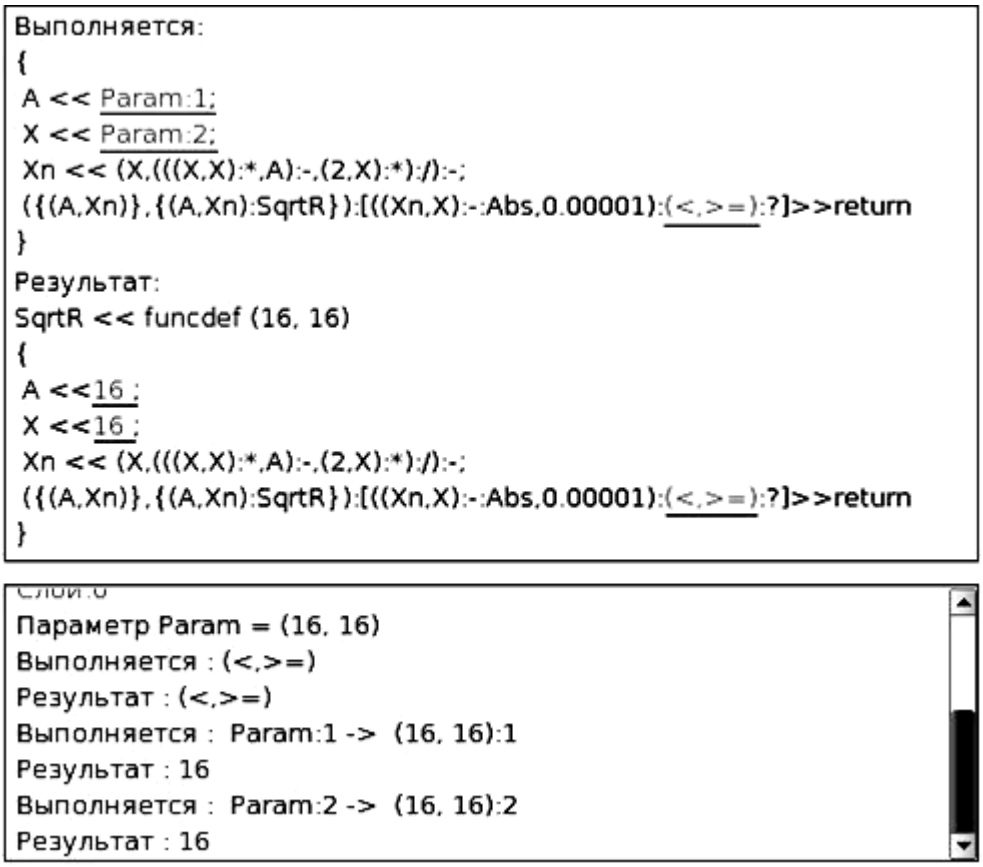


Рисунок А.19 – Текстовое представление первого шага отладки SqrtR

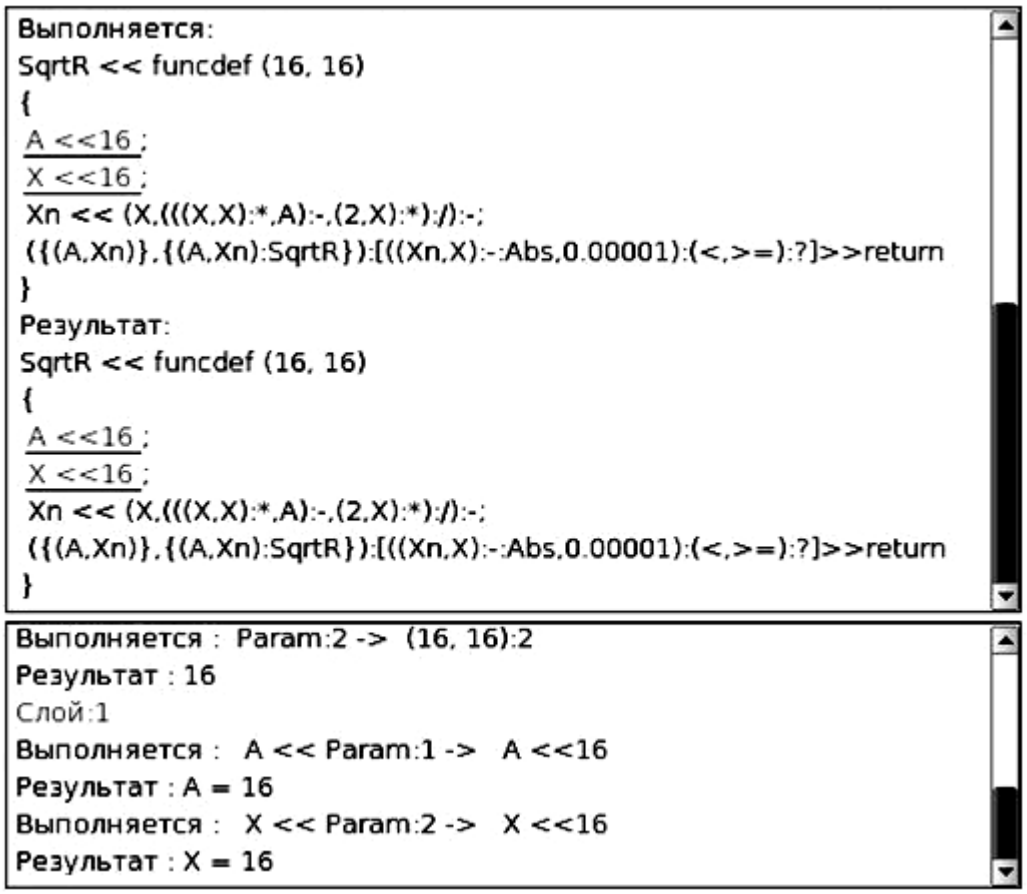


Рисунок А.20 – Текстовое представление второго шага отладки SqrtR

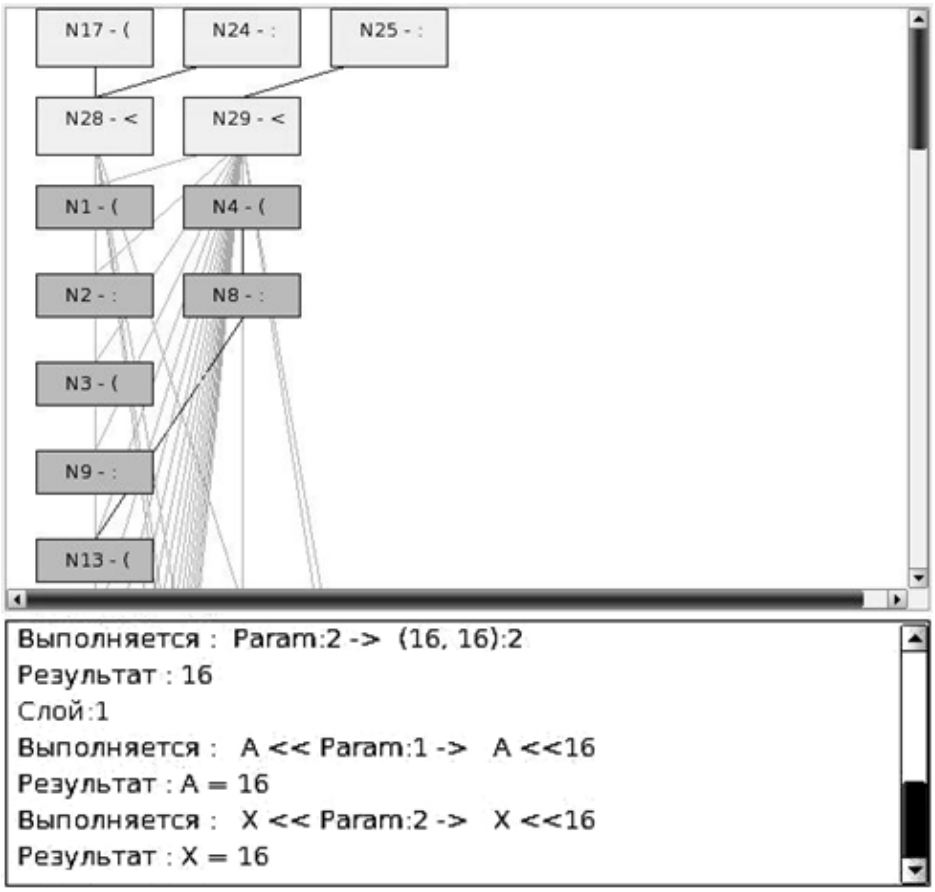


Рисунок А.21 – Графическое представление второго шага отладки SqrtR

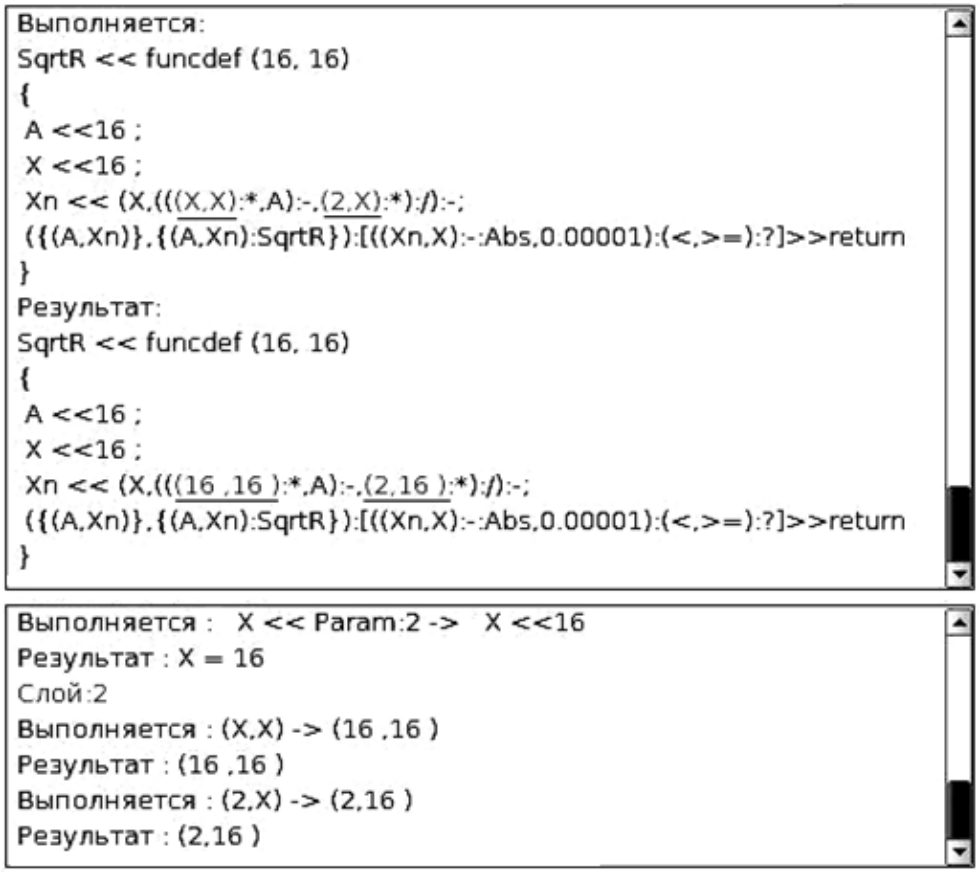


Рисунок А.22 – Текстовое представление третьего шага отладки SqrtR

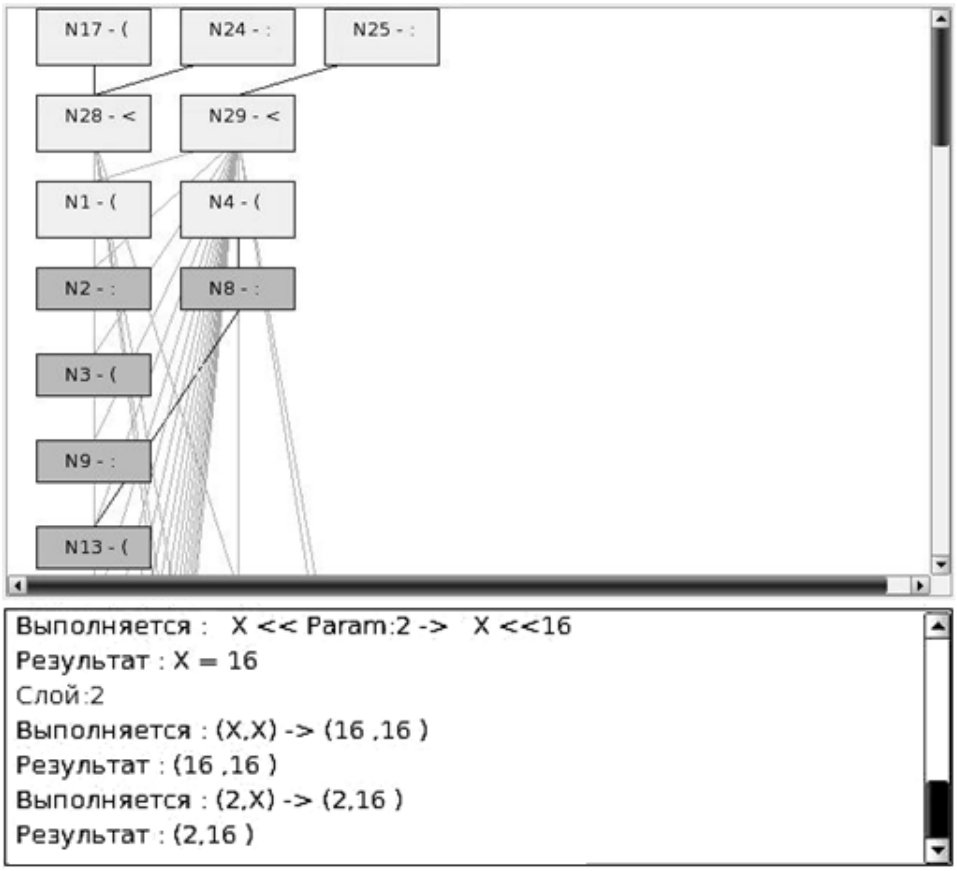


Рисунок А.23 – Графическое представление третьего шага отладки SqrtR

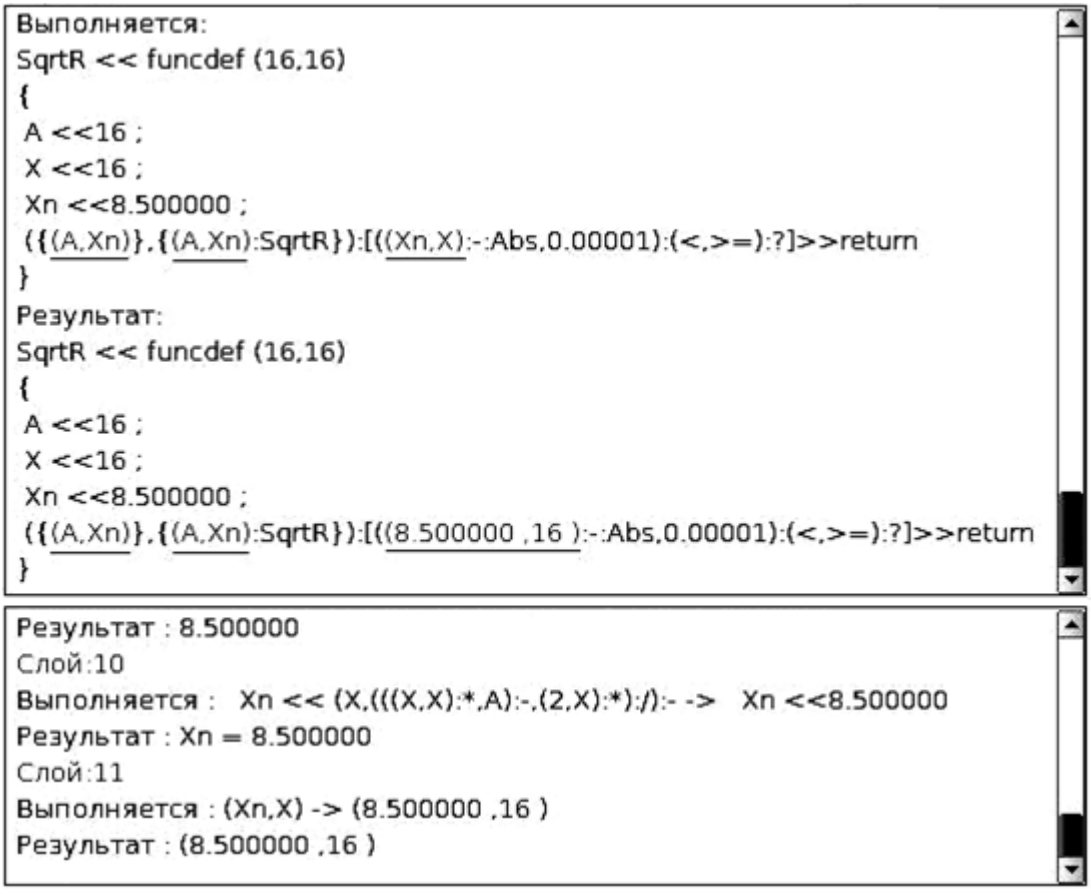


Рисунок А.24 – Текстовое представление двенадцатого шага отладки SqrtR

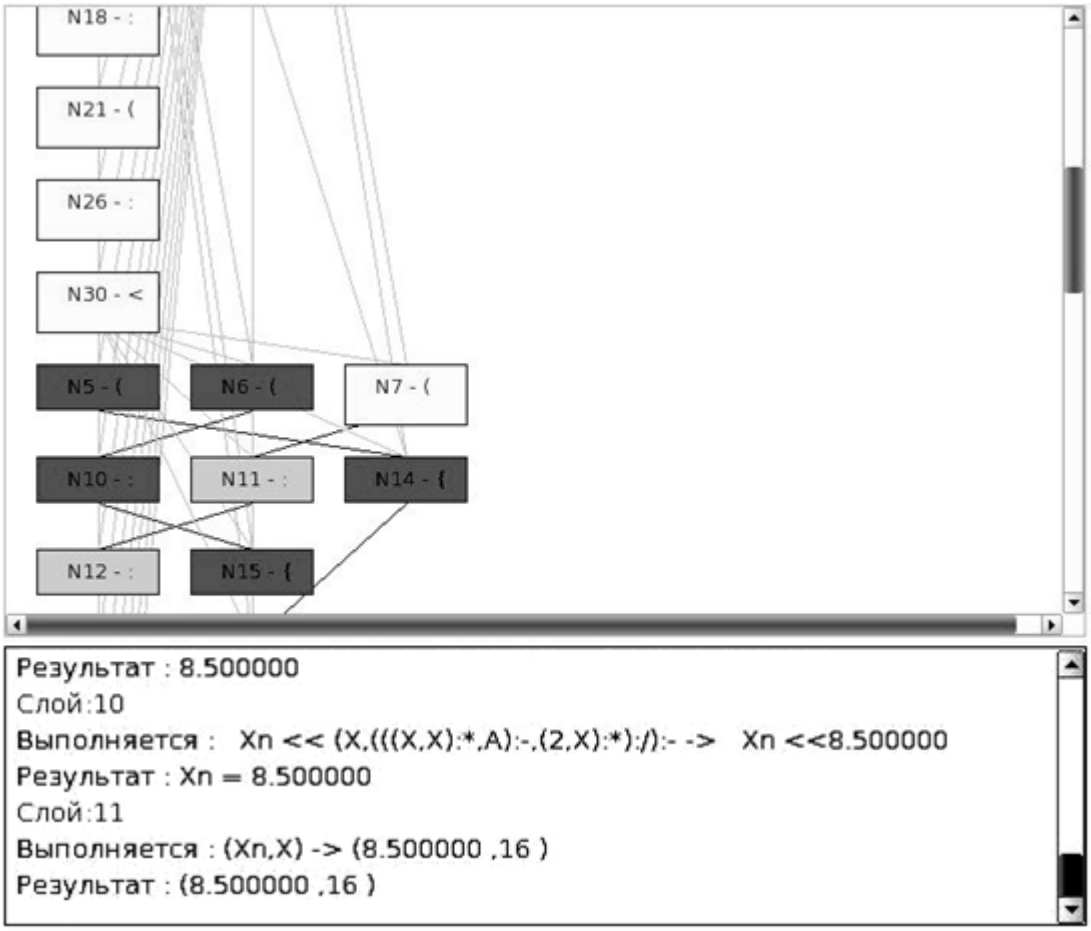


Рисунок А.25 – Графическое представление двенадцатого шага отладки SqrtR

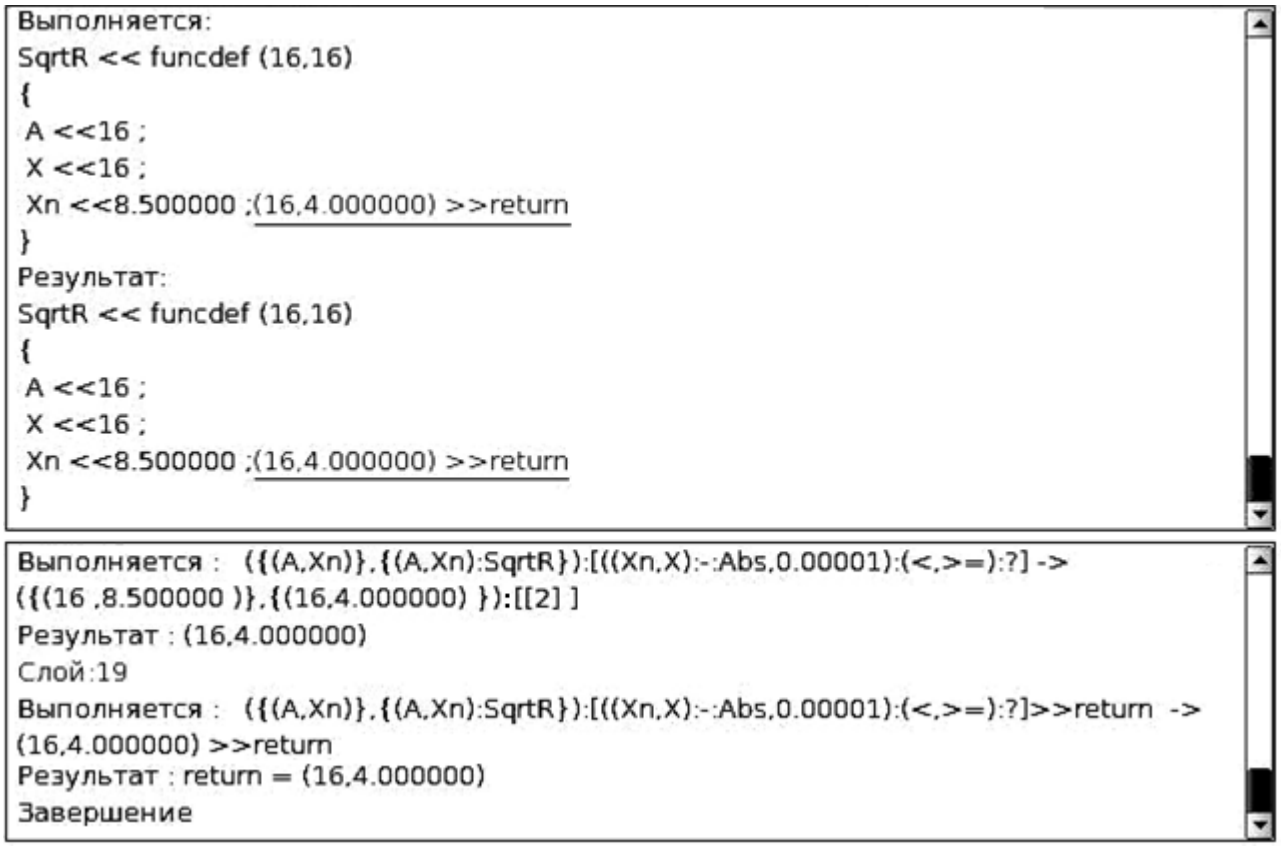


Рисунок А.26 – Текстовое представление последнего шага отладки SqrtR

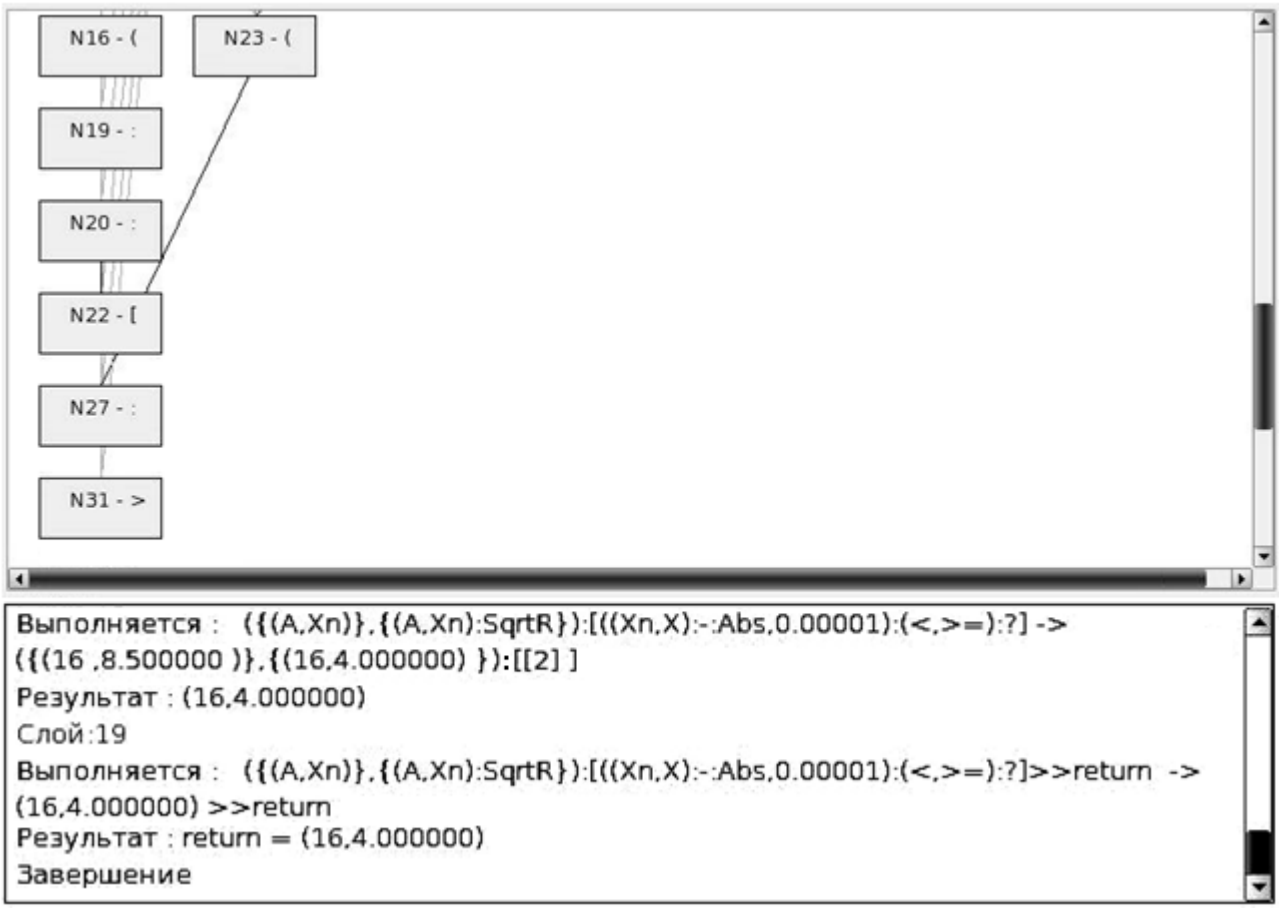


Рисунок А.27 – Графическое представление последнего шага отладки функции

На двенадцатом шаге отладки (рисунки А.24, А.25) слой состоит из трех операторов №5, №6, №7, два из которых являются задержанными. Так как была выбрана отладка с поддержкой задержанных вычислений, на шаге отладки будет выполнен только один оператор. Вычисление же двух оставшихся произойдет на некотором следующем шаге отладки, при необходимости получения их значений для выполнения не задержанного оператора. При подстановке значений операторов слоя в код функции, не вычисленные задержанные операторы останутся неизменными (рисунок А.24).

Приложение Б

Примеры обработки констант спецификации

В приложении представлены правила и примеры обработки верификатором операций интерпретации, хотя бы один аргумент которых задан формулой спецификации. Верификатор является программным модулем интегрированной среды разработки, отладки и верификации функционально-потоковых параллельных программ, архитектура которой описана в пятой главе.

Б.1 Функции над одним аргументом

~unknownnumber : +	~unknownnumber
~unknownbool : +	~unknownbool
~unknown : +	~unknown
~gt A : +	~gt A
~lt A : +	~lt A
~ge A : +	~ge A
~le A : +	~le A
~A interval B : +	~A interval B

~unknownnumber : -	~unknownnumber
~unknownbool : -	~unknownbool
~unknown : -	~unknown
~gt A : -	~lt -A
~lt A : -	~gt -A
~ge A : -	~le -A
~le A : -	~ge -A
~A interval B : -	~(-B) interval (-A)

~unknownnumber : *	Error
~unknownbool : *	~unknownbool
~unknown : *	~unknown
~gt A : *	Error
~lt A : *	Error
~ge A : *	Error
~le A : *	Error
~A interval B : *	Error

~unknownnumber : /	Error
~unknownbool : /	Error
~unknown : /	~unknown
~gt A : /	Error
~lt A : /	Error
~ge A : /	Error
~le A : /	Error
~A interval B : /	Error

Функция |, вычисляющая длину списка, не зависит от применяемых констант спецификации. Если значение не заключено в список, результатом будет ошибка:

~unknownnumber : |	Error
~unknownbool : |	Error
~unknown : |	~unknown
~gt A : |	Error
~lt A : |	Error
~ge A : |	Error
~le A : |	Error
~A interval B : |	Error

Если значения заключены в список, произойдет подсчет их числа, например:

(~A interval B) : |	1
(~unknownbool, ~unknown, ~le A) : |	3
((~unknownbool, ~unknown), 806) : |	2

~unknownnumber : %	Error
~unknownbool : %	Error
~unknown : %	~unknown
~gt A : %	Error
~lt A : %	Error
~ge A : %	Error
~le A : %	Error
~A interval B : %	Error

~unknownnumber : .	~unknownnumber
~unknownbool : .	~unknownbool
~unknown : .	~unknown
~gt A : .	~gt A
~lt A : .	~lt A
~ge A : .	~ge A
~le A : .	~le A
~A interval B : .	~A interval B

Если аргумент функции “.” заключен в последовательный () или параллельный список, результат также совпадает с аргументом. Примеры:

(~lt A) : .	(~lt A)
[~unknownbool] : .	[~unknownbool]

Если аргумент функции “.” заключен в задержанный список {}, результатом будет аналогичный параллельный список. Пример:

{~lt A} : .	[~lt A]

Если в списке аргументе функции более одного значения, результат будет аналогичным.

(~lt A, 0, 1) : .	(~lt A, 0, 1)
{~lt A, ~ge A, 1} : .	[~lt A, ~ge A, 1]
{~lt A, {~ge A, 1}} : .	[~lt A, ~ge A, 1]

~unknownnumber : ?	Error
~unknownbool : ?	~0 interval 1
~unknown : ?	~unknown
~gt A : ?	Error
~lt A : ?	Error
~ge A : ?	Error
~le A : ?	Error
~A interval B : ?	Error

Функции #, “..”, dup, “целое число” и все логические функции ==, !=, >=, <=, >, < при получении единственного аргумента, содержащего константу спецификации, возвратят ошибку.

Функции группировки в списки (), [], {}, создадут соответственные списки из одного аргумента. Пример:

~gt A : ()	(~gt A)

Б.2 Функции с двумя или более аргументами

Следующие константы спецификации, описывают интервал.

~unknownnumber	[-∞, +∞]
~gt A	(A, +∞]
~lt A	[-∞, A)
~ge A	[A, +∞]

~le A	[-∞, A]
~A interval B	[A, B]

Конкретное число тоже можно представить в виде интервала - C = [C, C].

Если арифметические функции (+, -, *, /) имеют в качестве обоих аргументов интервалы, заданные как две константы спецификации, или как число и константа спецификации, результат вычисляется по следующим правилам [134].

[a, b] + [c, d] = [a + c, b + d]

[a,b] - [c,d] = [a - d, b - c]

[a,b] * [c,d] = [min (a*c, a*d, b*c, b*d), max (a*c, a*d, b*c, b*d)]

[a,b] / [c,d] = [min (a/c, a/d, b/c, b/d), max (a/c, a/d, b/c, b/d)], если ноль не принадлежит интервалу [c,d].

При определении результата работы функции считается, что для любого известного положительного числа X, выполняется: +∞ (+, -, *, /) X = +∞, -∞ (+, -, *, /) X = -∞, +∞ (*, /) (-X) = -∞, -∞ (*, /) (-X) = +∞, а также +∞*+∞=+∞, -∞*+∞=-∞, -∞*-∞=+∞, 0*∞=0. И для любого числа X, выполняется: -∞<X, +∞>X, X/0=+∞, если X>0 и X/0=-∞, если X<0. Использование такого правила как 0*∞=0 является возможным, так как применение константы спецификации, описывающей бесконечный интервал, говорит об отсутствии информации относительно предполагаемого максимального или минимального значения величины, тем не менее, это значение не может быть бесконечным, при вычислении функции над конкретными данными.

Правила выше приведены для замкнутых интервалов. Для открытых интервалов они аналогичны. Но арифметическая операция может выполняться над замкнутым и открытым интервалом, в этом случае вид результирующего интервала определяется следующим образом. По вычислению границы интервала, проверяется из каких интервалов были взяты образующие ее операнды (пример: a*c, b/d), если хотя бы один операнд имеет открытую границу, то результатом также будет открытый или полуоткрытый интервал.

Отдельно следует сказать про операцию /. Если интервал [c,d] содержит ноль, то результат операции устанавливается в ~unknown и пользователь оповещается о возможной ошибке — возможном делении на ноль.

Пример выполнения арифметической функции:

(~lt -2, ~gt 5):* = [-∞,-2)*(5,+∞] = (min(-∞*5,-∞*+∞,-2*5,-2*+∞), max(-∞*5,-∞*+∞,-2*5,-2*+∞)) = (min(-∞,-∞,-10,-∞), max(-∞,-∞,-10,-∞)) = [-∞, -10) = ~lt -10

Функция % при работе с двумя интервалами будет вычисляться по следующему правилу – ([a,b], [c,d]):% обработается, если [c,d] не содержит нуля, и даст список из двух интервалов, где первый будет соответствовать целочисленному делению [a,b] / [c,d], а в качестве второго будет взят интервал [0, max (c, d, -c, -d)] если (a>0 и b>0 и c>0 и d>0) или (a>0 и b>0 и c<0 и d<0), [min (c, d, -c, -d), 0] если (a<0 и b<0 и c<0 и d<0) или (a<0 и b<0 и c>0 и d>0), [min (c, d, -c, -d), max (c, d, -c, -d)] в случае не выполнения ни одного из предыдущих условий.

Если аргументами функций +, -, *, / является число или интервал и логическая константа или ~unknownbool, то результатом будет Error. Если среди аргументов функций +, -, *, / есть константа спецификации ~unknown, то результат будет ~unknown. При работе с логическими значениями используются следующие правила:

(~unknownbool, ~unknownbool) : +	~unknownbool
(~unknownbool, true) : +	true
(~unknownbool, false) : +	~unknownbool
(~unknownbool, ~unknownbool) : *	~unknownbool
(~unknownbool, true) : *	~unknownbool
(~unknownbool, false) : *	false
(~unknownbool, ~unknownbool) : -	~unknownbool
(~unknownbool, true) : -	~unknownbool
(~unknownbool, false) : -	~unknownbool
(~unknownbool, ~unknownbool) : /	Error
(~unknownbool, true) : /	Error

(~unknownbool, false) : / Error

Здесь функция + понимается как логическое или, а * как логическое и. То есть и при большем числе аргументов, функция + будет возвращать true, если среди аргументов есть хотя бы один равный true, а функция * аналогично будет работать с false.

Логические функции ==, !=, >, <, >=, <= при работе с константами спецификации, описывающими интервал, определяют следующее: если нет такой пары чисел, взятой из интервалов, которая бы была ==, !=, >, <, >=, <=, то результатом работы функции является false, если такая пара чисел существует, то результат работы функции ~unknownbool, это означает, что функция может возвратить как ложь, так и истину. Значение true выдается в том случае, когда для любой пары чисел из интервалов выполняется отношение ==, !=, >, <, >=, <=. Так, для функции == и двух интервалов [a, b] и [c, d], если пересечение интервалов пустое, то есть b<c, то функция возвратит false, иначе ~unknownbool. Значение true возможно в случае совпадающих вырожденных интервалов.

Функция >, работающая с двумя интервалами [a, b] и [c, d], возвратит false, если b<c, true, если a>d, и ~unknownbool если первые два условия не выполняются. Функция <, работающая с двумя интервалами [a, b] и [c, d], возвратит false, если a>d, true, если b<c, и ~unknownbool если первые два условия не выполняются.

Для логических аргументов верны следующие соответствия.

(~unknownbool, ~unknownbool) : == ~unknownbool

(~unknownbool, true) : == ~unknownbool

(~unknownbool, false) : == ~unknownbool

(~unknownbool, ~unknownbool) : != ~unknownbool

(~unknownbool, true) : != ~unknownbool

(~unknownbool, false) : != ~unknownbool

(~unknownbool, ~unknownbool) : >= ~unknownbool

(~unknownbool, true) : >= ~unknownbool

(~unknownbool, false) : >=	true
(~unknownbool, ~unknownbool) : <=	~unknownbool
(~unknownbool, true) : <=	true
(~unknownbool, false) : <=	~unknownbool
(~unknownbool, ~unknownbool) : >	~unknownbool
(~unknownbool, true) : >	false
(~unknownbool, false) : >	~unknownbool
(~unknownbool, ~unknownbool) : <	~unknownbool
(~unknownbool, true) : <	~unknownbool
(~unknownbool, false) : <	false

Функция ? выполняемая над списком возвратит ошибку, если среди элементов списка есть хотя бы одна интервальная константа спецификации или число. Функция возвращает номера тех элементов списка, которые равны true. Если среди элементов списка присутствует хотя бы одна константа ~unknownbool, то существует несколько вариантов вычисления функции ?. Пример: (true, ~unknownbool, ~unknownbool):? может сформировать следующие результаты [1], [1,2,3], [1,2], [1,3]. Все варианты выполнения функции ? вычисляются и предоставляются пользователю, для выбора дальнейшего пути вычисления.

Функция # преобразует список и не зависит от применяемых констант спецификации. Пример:

((false, true), (~ge6, 7)):#	((false, ~ge6), (true, 7))

Функция число изымает элемент из списка, она также будет работать одинаково для любых применяемых констант спецификации. Примеры:

(0, (0, ~le1), ~unknown):3	~unknown
(0, (0, ~le1), ~unknown):2	(0, ~le1)
(0, (0, ~le1), ~unknown):4	Error
(0, (0, ~le1), ~unknown):-2	(0, ~unknown)

Более сложным является случай, когда функция сама задана константой спецификации, описывающей интервал. Здесь, как и для функции ? возможно вычислить все возможные результаты ее выполнения. Например: (0, (0, ~le1), ~unknown):~1 interval 10 может дать такие результаты как 0, (0, ~le1), ~unknown, Error. Значение Error будет получено не только при выходе за границы списка, но и при задании в качестве функции дробного числа.

Функции .. и dup при получении константы спецификации, описывающей интервал, будут иметь несколько вариантов выполнения. Функция dup получает список из двух аргументов, несколько вариантов исполнения возникнут, если второй аргумент является интервалом. Результатом (~ge4, 3):dup будет (~ge4, ~ge4, ~ge4), тогда как для (3, ~le4) :dup возможны следующие значения (3, 3, 3, 3), (3, 3, 3), (3, 3), (3), (), Error. Возникновение ошибки может быть связано как с отрицательным, так и дробным значением. Здесь существует отличие от функций ? и #, если второй аргумент является интервалом, включающим $+\infty$, то не возможно предоставить пользователю все возможные варианты исполнения. В этом случае отправляется запрос к пользователю о предполагаемом результате выполнения функции.

Приложение В

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2014610559

Верификатор функционально-потоковых параллельных программ на языке Пифагор под ОС Linux

Правообладатель: ***Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Сибирский федеральный университет» (СФУ) (RU)***

Авторы: ***Легалов Александр Иванович (RU),***
Удалова Юлия Васильевна (RU)

Заявка № **2013660557**
Дата поступления **18 ноября 2013 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***15 января 2014 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Б.П. Симонов

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2012612190

Графический отладчик функционально-потоковых параллельных программ на языке Пифагор под ОС Linux

Правообладатель(ли): ***Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Сибирский федеральный университет» (СФУ) (RU)***

Автор(ы): ***Легалов Александр Иванович, Непомнящий Олег Владимирович, Удалова Юлия Васильевна, Хабаров Виталий Александрович (RU)***

Заявка № **2011660167**

Дата поступления **28 декабря 2011 г.**

Зарегистрировано в Реестре программ для ЭВМ

28 февраля 2012 г.

Руководитель Федеральной службы по интеллектуальной собственности

Б.П. Симонов

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное автономное образовательное
учреждение высшего профессионального образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

SIBERIAN FEDERAL UNIVERSITY
СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

660041, Россия, Красноярск, проспект Свободный, 79
телефон (391) 244-82-13, факс (391) 244-86-25
http://www.sfu-kras.ru e-mail: office@sfu-kras.ru

№ _____
на № _____ от 20.12.2013

Справка

Подтверждаю участие соискателя Удаловой Юлии Васильевны в 2012-2013 годах в качестве исполнителя в научно-методическом проекте «Инструментальная поддержка архитектурно-независимой разработки параллельных программ на основе функционально-потоковой парадигмы параллельного программирования» в рамках федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» (ФЦП-6).

Начальник НИЧ СФУ С.В. Первухин

Главный специалист НИЧ СФУ В.А. Пригожих

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное автономное образовательное
учреждение высшего профессионального образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

SIBERIAN FEDERAL UNIVERSITY
СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

660041, Россия, Красноярск, проспект Свободный, 79
телефон (391) 244-82-13, факс (391) 244-86-25
http://www.sfu-kras.ru e-mail: office@sfu-kras.ru

№ _____
на № _____ от 20.12.2013

Справка

Подтверждаю участие соискателя Удаловой Юлии Васильевны в 2007-2008 годах в качестве исполнителя в научно-методическом проекте №25 «Среда разработки для языка параллельного программирования Пифагор» в рамках «Программы развития СФУ на 2007–2010 годы».

Начальник НИЧ СФУ _____ /Первухин С.В./

Начальник НИС СФУ _____ /Темных В.И./

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное автономное образовательное
учреждение высшего профессионального образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

SIBERIAN FEDERAL UNIVERSITY
СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

660041, Россия, Красноярск, проспект Свободный, 79
телефон (391) 244-82-13, факс (391) 244-86-25
http://www.sfu-kras.ru e-mail: office@sfu-kras.ru

№ _____

на № _____ от _____

Утверждаю
Проректор
по учебной работе
_____ Н.В. Гафурова
«7» июня 2012 г.

СПРАВКА
о внедрении результатов диссертационных исследований в учебный процесс

Комиссия в составе: **председатель- доцент, начальник инновационно-методического управления СФУ Смолин А.Ю.**
члены комиссии: **д.т.н., проф. , директор ИКИТ, зав. кафедрой СИИ Цибульский Г.М.,**
к.т.н., зав. каф. ВТ, Середкин В.Г.,
д.т.н., проф. Краснобаев Ю.В.

составили настоящую справку о том, что результаты научных исследований, проведенных Удаловой Юлией Васильевной в рамках диссертационной работы на тему «Отладка и верификация функционально-потоковых параллельных программ», внедрены в учебный процесс по курсу «Технология программирования» в виде лекций по теме жизненный цикл процесса разработки программного обеспечения. Результаты исследований используются для подготовки специалистов по специальности 230101.65 – «Вычислительные машины, комплексы, системы и сети».

Председатель комиссии _____ Смолин А.Ю.

Члены комиссии _____ Цибульский Г.М.

_____ Середкин В.Г.

_____ Краснобаев Ю.В.