Journal of Siberian Federal University. Engineering & Technologies 2 (2011 4) 213-224
~ ~ ~

УДК 681.3.06

Методы отладки и верификации функционально-потоковых параллельных программ

Ю.В. Удалова,
А.И. Легалов, Н.Ю. Сиротинина
Сибирский федеральный университет,
Россия 660041, Красноярск, пр. Свободный, 79 [1]

Received 5.04.2011, received in revised form 12.04.2011, accepted 19.04.2011

В статье предлагаются методы отладки функционально-потоковых параллельных программ, обеспечивающие гибкий анализ параллельных процессов и их визуальное восприятие с различных позиций. Рассмотрен подход к их формальной верификации, базирующийся на переборе, поступающих в асинхронные списки данных, эквивалентный методу проверки моделей. Это позволяет проанализировать корректность разрабатываемых программ без построения специализированных промежуточных моделей.

Ключевые слова: отладка, верификация, параллельные вычисления, функционально-потоковое параллельное программирование.

Введение

Отладка и верификация являются важными этапами процесса разработки программного обеспечения. Задача отладки, состоящая в обнаружении и локализации ошибок, достаточно успешно решается для последовательного программирования. Существующие отладчики позволяют использовать разнообразные приемы и инструменты, например пропуск уже отлаженных фрагментов программ, отладку только требуемых функций или процедур, удобный просмотр уже отлаженного кода, пошаговую отладку и т.д. Отладчики функциональных программ также предоставляют обширный инструментарий, позволяющий задавать и производить шаг отладки, устанавливать контрольные точки, выполнять отладку частично незаконченных программ. При переходе к параллельным вычислениям процесс отладки сопровождается дополнительными трудностями [1,2], обуславливаемыми спецификой взаимодействия процессов. В частности, необходимо анализировать конфликтные и тупиковые ситуации, возникающие при взаимодействии процессов и потоков, как через общую память, так и через каналы передачи сообщений. В настоящее время методы отладки параллельных программ постоянно совершенствуются. Достигнуты определенные успехи в разработке инструментов для систем, использующих как потоки, так и процессы [3-6].

* Corresponding author E-mail address: uuuu82@inbox.ru

Верификация – процесс, направленный на определение наличия ошибок и условий их возникновения. Развиваются методы формальной верификации параллельных программ, обеспечивающие дополнительный анализ корректности взаимодействия процессов. Существенно продвинулись методы формальной верификации, базирующиеся на проверке моделей [1,7]. Если отладка программы направлена на выборочную подстановку данных и не доказывает правильность программы, то верификация позволяет анализировать свойства последовательной программы по ее тексту [8-11] или обеспечивает перебор всех возможных путей выполнения параллельной программы [1,7,12,13]; известны методы формальной верификации параллельных программ [14].

Используемые в настоящее время способы создания прикладного параллельного программного обеспечения порождают ряд проблем, связанных с масштабируемостью и переносимостью программ, эффективностью как самого программного обеспечения, так и процесса его разработки. Поэтому разрабатываются альтернативные подходы, повышающие эффективность параллельного программирования. В частности, предлагаемая авторами функционально-потоковая парадигма программирования обеспечивает описание ресурсно-независимого параллелизма [15,16]. На основе этой парадигмы разработан язык программирования Пифагор [17], позволяющий писать архитектурно-независимые параллельные программы. Методы отладки и верификации функционально-потоковых параллельных программ [18,19] дают возможность использовать ряд специфических приемов, определяемых особенностью модели вычислений, в которой отсутствует понятие переменной, выполняемые функции связаны напрямую, а данные имеют динамическую типизацию.

Целью работы является разработка новых методов отладки и формальной верификации, расширяющих инструментальную поддержку функционально-потокового параллельного программирования. Повышение эффективности процесса обнаружения ошибок в программе за счет более детального и наглядного представления данных, использование методов формальной верификации позволяют повысить надежность функционально-потоковых параллельных программ.

Методы отладки функционально-потоковых параллельных программ

Модель вычислений определяет функционально-потоковую параллельную программу как информационный граф, в котором вершины служат операторами, а дуги – 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. Отладка функционально-потоковой параллельной программы может проходить в одном из четырех режимов: пошаговой отладки, отладки слоев, отладки ветвей и проверки формул.

Режим пошаговой отладки функционально-потоковой параллельной программы подобен последовательному режиму отладки. Возможность его использования 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

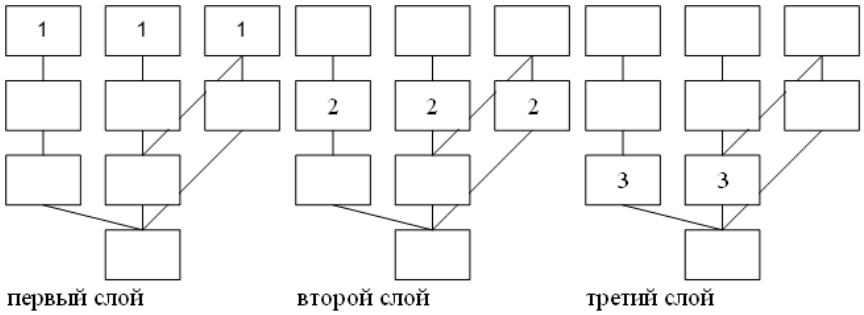

Рис. 1. Пример послойного выполнения операторов функционально-потоковой параллельной программы

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 к другой не приводит к противоречиям с непосредственным выполнением программы, на каком бы уровне вложенности функций он ни произошел.

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 (рис. 1). После того, как эти операторы окажутся вычисленными, в графе программы соответствующие 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.

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 не зависимы 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. Под ошибкой здесь понимается специальная константа, обозначающая результат такой операции, как деление на ноль, выход за границы списка, арифметическое действие над числом и символом и т.п. 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.

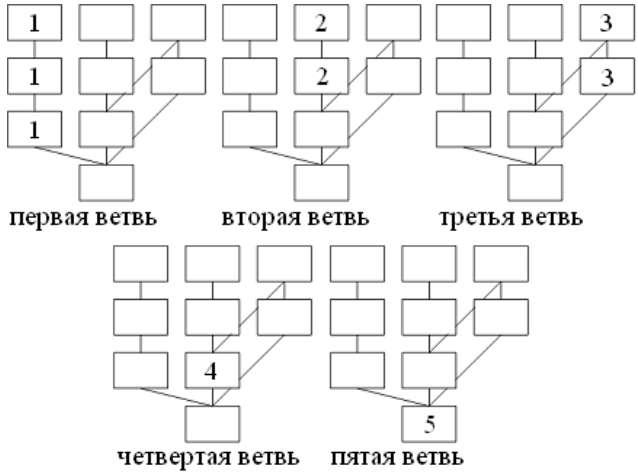

Рис. 2. Использование ветвей при отладке операторов функционально-потоковой параллельной программы

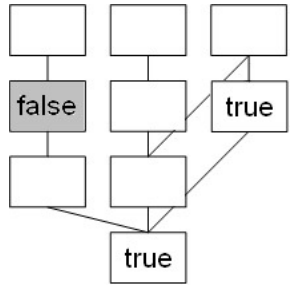

Рис. 3. Разметка вершин при отладке в режиме проверки формул

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.

Режим отладки ветвей (рис. 2) предполагает, что на каждом шаге отладки для выполнения выбираются все операторы независимой ветви информационного графа. 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.

Режим проверки формул (рис. 3) позволяет пользователю закрепить за произвольными узлами-операторами собственные утверждения (являющиеся выражениями на языке программирования Пифагор), использующие в качестве значений аргумент функции или значения,

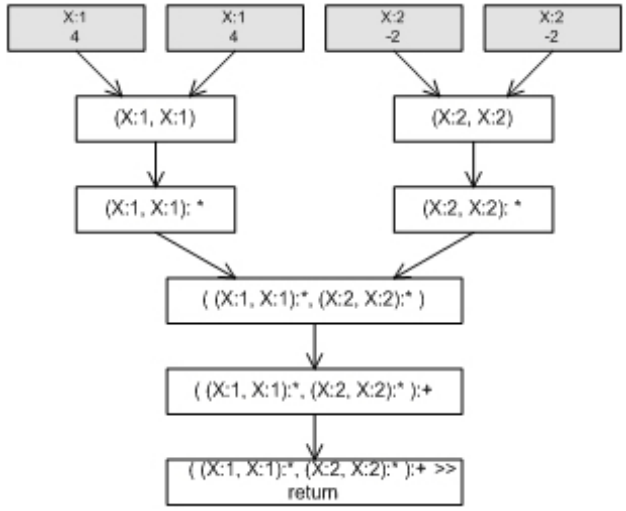

Рис. 4. Первый шаг отладки

вычисленные любым узлом-оператором графа. Если отлаживаемая функция не рекурсивная, отладка выполняется за один шаг, вычисляются все узлы графа и все дополнительные утверждения.

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.

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 (рис. 4).

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**;

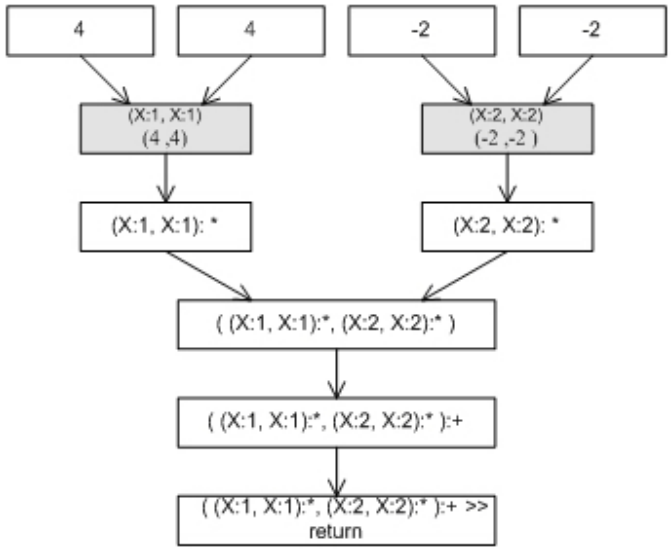

Рис. 5. Второй шаг отладки

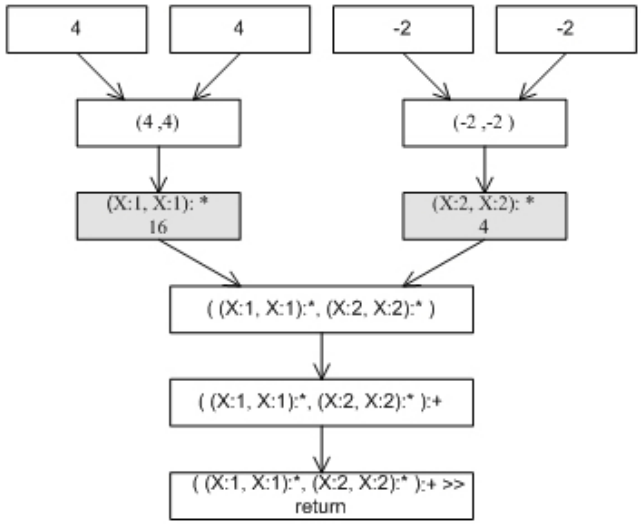

Рис. 6. Третий шаг отладки

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;}

Выполнение операторов первого слоя на втором шаге отладки (рис. 5) 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):

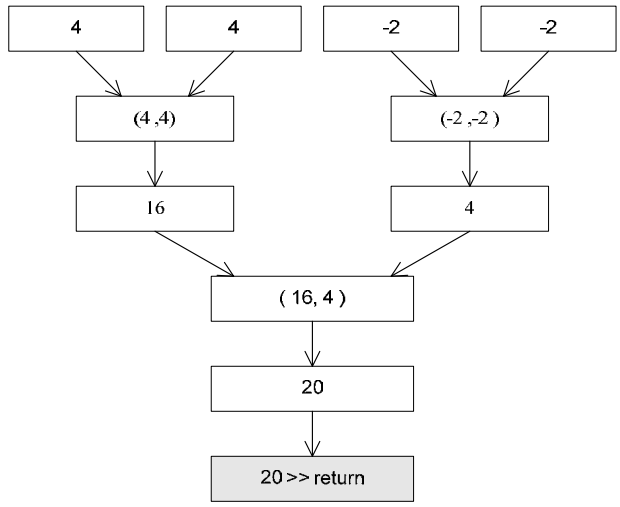

Рис. 7. Последний шаг отладки

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

Следующий слой связан с выполнением двух операций умножения (рис. 6):

оператор **(x:1,x:1):*** реализует **(4, 4):*** и дает **16**;

оператор **(x:2,x:2):*** реализует **(-2, -2):*** и дает **4**.

На третьем шаге (рис. 6) в отлаживаемой функции произойдет выделение цветом сформированных списков (что отображено ниже курсивом с подчеркиванием):

summa << funcdef (4,-2) { (*16*, *4*):+ >> return;}

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, изображенному на рис. 7.

Функция, после завершения отладки, выглядит следующим образом:

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

Пример использования отладчика в режиме проверки формул. Рассмотрим отладку этой же функции в режиме проверки формул. Добавим требования к трем узлам графа (рис. 8), где **(NODE,0):>** означает, что значение вершины-оператора больше нуля, и **(NODE,0):=** означает, что значение вершины-оператора равно нулю.

Отладка проходит за один шаг (рис. 9), на котором вычисляются значения операторов и выражений, приписанных к узлам. В зависимости от значений выражений расцвечиваются узлы графа, по нажатию на вершину с приписанными условиями можно увидеть условия и их значения; так, для результирующего узла графа **(NODE,0):> → (20,0):> → true**.

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.

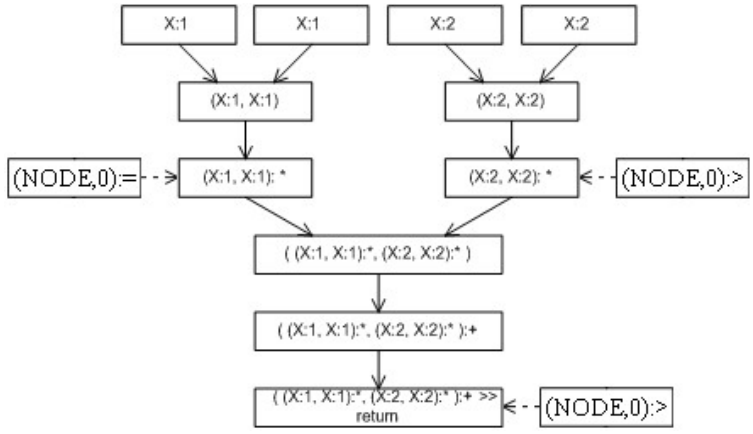

Рис.8. Условия пользователя в режиме проверки формул

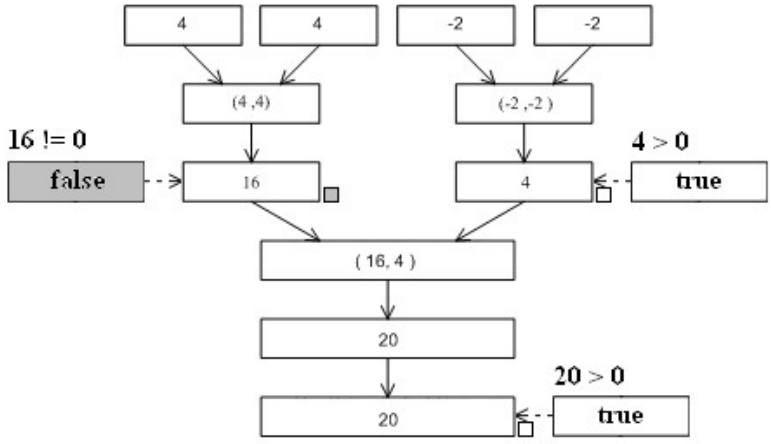

Рис. 9. Вычисление условий в режиме проверки формул

Метод верификации функционально-потоковых параллельных программ, использующих асинхронные списки

Сложность верификации параллельных программ заключается в огромном количестве вариантов последовательности исполнения операторов, в которой скрываются не только логические ошибки, но и ошибки, связанные с параллельным исполнением фрагментов программ: ошибки взаимодействия и синхронизации; конфликты, возникающие при одновременном обращении нескольких процессов к одним и тем же общим ресурсам и т. п.

Модель функционально-потоковых параллельных вычислений [15-17] исключает ошибки взаимодействия и синхронизации, а также зависимость результатов от порядка выполнения операторов, готовых к вычислению. Исключение составляет ситуация, возникающая при использовании асинхронных списков [20]. 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 спи-

ска 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 такой 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 вычисляют все его элементы, потом производят 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.

Рассмотрим рекурсивную функцию, получающую 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-…

```
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
// проверяем 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; }
```

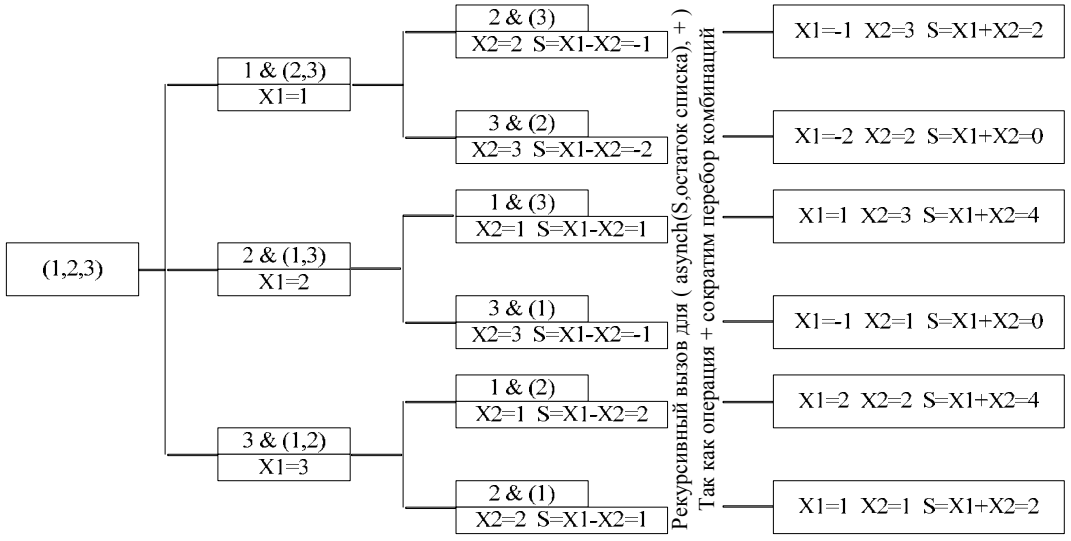

Рис. 10. Сокращенные пути выполнения асинхронной программы

В качестве начальных данных укажем список вида **(asynch(1,2,3), -)**. Рассмотрим все возможные пути выполнения асинхронной программы (рис. 10).

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

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

Заключение

Использование функционально-потоковой парадигмы программирования вместо традиционного императивного подхода позволяет избавиться при разработке параллельных программ от ряда ошибок, обуславливаемых ресурсными конфликтами, возникающими между взаимодействующими процессами. Вместе с тем, это не ведет к исчезновению логических ошибок и ряду ошибок, возникающих при более сложном взаимодействии параллельных процессов. Предлагаемые в работе методы отладки и верификации обеспечивают более наглядное, по сравнению с уже существующими отладчиками, представление отладочных данных и позволяют быстрее обнаруживать и исправлять имеющиеся ошибки.

Формальная верификация функционально-потоковых параллельным 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.

Статья подготовлена и публикуется при поддержке Программы развития Сибирского федерального университета.

Список литературы

1. Карпов, Ю.Г. MODEL CHECKING. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010. 560 с.

2. Information Flow and Usage in an E-shop Operating within an Agent-based E-commerce System / Drozdowicz M., Ganzha M. Paprzycki M., Gawinecki M., Legalov A. // Журнал Сибирского федерального университета. 2009. №1. С. 3-22.

3. Калинов, А.Я., Карганов, К.А., Хоренко, К.В. Команда «шаг» в параллельных отладчиках// Труды Института системного программирования РАН Вып 5 – 2004. – С. 89-101.

4. Отладчик Panorama. [Электронный ресурс] – Режим доступа: http://www.cs.ucsd.edu/users/berman/panorama – Загл. с экрана.

5. Отладчик TotalView. [Электронный ресурс] – Режим доступа: http://www.totalviewtech.com/products/totalview.htm – Загл. с экрана.

6. Отладчик Ozcar. [Электронный ресурс] – Режим доступа: http://www.mozart-oz.org/documentation/ozcar/index.html – Загл. с экрана.

7. Кларк, Э.М., Грамберг, О., Пелед, Д. Верификация моделей программ. М.: МЦНМО, 2002. 416 с.

8. Грис, Д. Наука программирования. М.: Мир, 1984. 416 с.

9. Лисков, Б., Гатэг, Дж. Использование абстракций и спецификаций при разработке программ. М.: Мир, 1989. 424 с.

10. Верификатор PVS. [Электронный ресурс] – Режим доступа: http://pvs.csl.sri.com – Загл. с экрана.

11. Верификатор Isabelle. [Электронный ресурс] – Режим доступа: http://www.cl.cam.ac.uk/research/hvg/Isabelle – Загл. с экрана.

12. Верификатор Spin. [Электронный ресурс] – Режим доступа: http://www.netlib.bell-labs.com/netlib/spin – Загл. с экрана.

13. Верификатор Bogor. [Электронный ресурс] – Режим доступа: http://bogor.projects.cis.ksu.edu – Загл. с экрана.

14. Непомнящий, В.А., Рякин, О.М. Прикладные методы верификации программ. М: Радио и связь, 1988. 255 с.

15. Модель параллельных вычислений функционального языка// Известия ГЭТУ. Сборник научных трудов. Вып. 500. Структуры и математическое обеспечение специализированных средств. СПб., 1996. С. 56-63.

16. Легалов, А.И. Об управлении вычислениями в параллельных системах и языках программирования// Научный вестник НГТУ. 2004. № 3 (18)С. 63-72.

17. Легалов, А.И. Функциональный язык для создания архитектурно-независимых параллельных программ// Вычислительные технологии. 2005. № 1 (10). С. 71-89.

18. Удалова Ю.В., Легалов, А.И., Сиротинина, Н.Ю. Средства отладки функционально-потоковых параллельных программ / Доклады АН ВШ РФ. 2008. № 1 (10). С. 96-105.

19. Удалова, Ю.В. Об отладке и верификации функционально-потоковых параллельных программ. [Электронный ресурс] / Ю.В. Удалова, А.И. Легалов, Н.Ю. Сиротинина, М.С. Кропачева // Параллельные вычислительные технологии (ПаВТ'2009): Труды международной на-

учной конференции (Нижний Новгород, 30 марта – 3 апреля 2009 г.). –Челябинск: Изд. ЮУрГУ, 2009. – С. 757-764. – 1 электрон. опт. диск (CD–ROM).

20. Легалов, А.И. Использование асинхронных вычислений в функциональных языках параллельного программирования // Распределенные и кластерные вычисления: Избранные материалы четвертой школы-семинара. Красноярск, 2005. С. 172-183.

Debug and Verification of Function-Stream Parallel Programs

Julia V. Udalova,
Alexander I. Legalov and Natalie U. Sirotinina
Siberian Federal University,
79 Svobodny, Krasnoyarsk 660041 Russia

Debug function-stream parallel program can pass in one of the four realized modes: mode of incremental debug, mode of layer debug, mode of branch debug and mode of formula's checking. Verification of function-stream parallel programs with asynchronous lists is described in article.

Keywords: debug, verification, function-stream parallel programs.