AUTOMATIC TEST GENERATION FOR INTEL GFX-OFFLOAD COMPILER

S. B. Pankratov
A. P. Ershov Institute of Informatics Systems (IIS),
Siberian Branch of the Russian Academy of Sciences
630090, Novosibirsk, Russia

High-level programming languages are the main development tool for software. The task of translating the source code of programs into a representation executable on a computer system is solved by the compiler. Compilers have high quality requirements since correctness of compiled programs significantly depends on compiler correctness. And software defects caused by errors in the compiler are difficult to identify, and impossible to correct without interference in the compiler itself. The correctness of the compiler is a necessary requirement for the correct operation of the software compiled by it. Therefore, the most important stage in the development of the compiler is verification.

Due to complexity of programs as compiler input data and their transformations, the task of compiler verification is very tedious and difficult. And in the case of using an optimizing compiler, it is also algorithmically unsolvable. But we can consider the behavior of the compiler on some limited class of programs. We cannot compare the result of the transformation with some reference, we can only consider some properties of the translation produced. For example, if the compilation of the same program without optimizations succeeds, and with optimizations leads fails in compiler, then we can talk about the presence of errors in the optimization code.

Recent widespread of multicore processors and graphics core integrated to CPU emphasized the problem of the transition from single-threaded to multithreaded computing and re-usage of graphics core for general purpose heterogeneous parallel computations in particular. Intel introduced GFX-offload extension for C++ language in addition to the some open standards for parallel computing: OpenMP, OpenCL, Cilk Plus. That helps to utilize the powers of integrated graphics for general purpose computation. Taking all these facts into account, verification of implementations of parallel C++ extension in compiler is particularly important.

Automatic test generation is an important part of the verification, because tests written by hand usually can't effectively cover all possible combinations of language constructions, as well as all situations of optimization's application. The QA command of the Intel compiler chose an approach using a parametric context-free grammar, described by a special meta-language for test generator. Parametric context-free grammars have proven themselves as a good formalism for constructing generators of semantically correct and compiler-specific tests with deterministic behavior. They were distinguished by the clarity of the description of generated test programs, as well as the flexibility and convenience in working with their context. However, this approach was used only to generate single-threaded tests executed on the central processor. Therefore it is an open question about the possibility of using an already existing generator of single-threaded tests based on parametric context-free grammars, for generating tests for multi-threaded extensions of the C ++ language, like GFX-offload.

In this paper, we presenting an approach to automate test creation for the verification of the GFX-offload compiler, based on a generator that uses grammars to generate syntactically correct executable tests. Also we will show some results of using this approach for compiler's verification at Intel Corporation.

Key words: QA, compilers, offload, test generation, GPU, grammar.

References

1. Stasenko A. P. Generation of executable tests for compiler // *Parallel programs construction and optimization.* 2008. 16. P. 301–313,

2. Aho A. V., Sethi R., Ullman J. D. Compilers: principles, techniques, and tools. Boston: Addison-Wesley Longman Publishing Co., Inc., 1986. P. 796.

3. Hanford K. V. Automatic generation of test cases // IBM Systems Journal. NY: IBM, Dec. 1970. Vol. 9. P. 242–257.

4. Purdom P. A sentence generator for testing parsers // Behavior and Information Technology, July 1972. Vol. 12. N 3. P. 366–375.

5. Hutchison J. S., Duncan A. G. Using Attributed Grammars to Test Designs and Implementation // In Proc. of the 5th international conference on Software engineering, 1981. P. 170–178.

6. Bazzichi F, Spadafora I. An automatic generator for compiler testing // IEEE transactions on Software Engineering. NY: IEEE, 1982. Vol. SE-8. P. 343 –353.

7. Kossatchev A. S., Posypkin M. A. Survey of compiler testing methods // Programming and Computing Software. NY: Plenum Press, 2005. Vol. 31, N 1. P. 10–19.

АВТОМАТИЧЕСКАЯ ГЕНЕРАЦИЯ ТЕСТОВ ДЛЯ GFX-OFFLOAD КОМПИЛЯТОРА INTEL

С. Б. Панкратов
Институт систем информатики им. А. П. Ершова СО РАН
630090, Новосибирск, Россия

УДК 004.43

Компилятор — инструмент, требования к надежности которого чрезвычайно высоки. Так как дефекты программного обеспечения, вызванные ошибками в компиляторе, сложно выявить, а тем более исправить без вмешательства в сам компилятор, поэтому важнейшим этапом разработки компилятора является его верификация. Из-за сложности входных данных и производимых над ними преобразований задача верификации компиляторов является весьма трудоемкой и непростой. А в случае использования оптимизирующего компилятора еще и алгоритмически неразрешимой, поэтому можем рассмотреть поведение компилятора только на некотором ограниченном классе программ. В статье представлен подход к автоматизации создания тестов для верификации GFX-offload компилятора, основанный на генераторе, использующем грамматики для порождения синтаксически корректных исполняемых тестов. Также приведены результаты использования полученной грамматики в процессе тестирования компилятора в компании Intel.

Ключевые слова: тестирование, генерация тестов, компиляторы, оффлоад, грамматики, графические ускорители.

Введение. Высокоуровневые языки программирования являются основным средством разработки программного обеспечения. Задача по переводу исходного текста программ в представление, выполнимое на вычислительной системе, решается компилятором. Компилятор — инструмент, требования к надежности которого чрезвычайно высоки. А дефекты программного обеспечения, вызванные ошибками в компиляторе, сложно выявить, а тем более исправить без вмешательства в сам компилятор. Корректность компилятора является необходимым требованием корректной работы программного обеспечения, собранного при его использовании. Поэтому важнейшим этапом разработки компилятора является его верификация.

Из-за сложности входных данных и производимых над ними преобразований задача верификации компиляторов является весьма трудоемкой и непростой. А в случае использования оптимизирующего компилятора еще и алгоритмически неразрешимой. Но мы можем рассмотреть поведение компилятора на некотором ограниченном классе программ. Мы не можем сравнить результат преобразования с некоторым эталонным, мы можем рассматривать только некоторые свойства произведенного перевода. Например, если компиляция одной и той же программы без оптимизаций завершается успешно, а с оптимизациями приводит к падению компилятора, то мы можем говорить о наличии ошибок в оптимизацирующем коде.

Работа выполнена при частичной финансовой поддержке Российского фонда фундаментальных исследований (грант РФФИ № 15-07-02029)

С появлением многоядерных процессоров острее встала проблема перехода от однопоточных к многопоточным вычислениям. А с момента переноса графического ядра на один кристалл с вычислительным и увеличением его мощности появилась идея по утилизации и его вычислительных мощностей в дополнение к процессорным. Так появилось параллельное GFX-offload расширение для языка C++ , позволяющее использовать графическое ядро как процессор общего назначения. Принимая все эти факты во внимание, тестирование реализаций параллельных расширений языка C++ становится очень важным дополнением к традиционному процессу верификации.

Автоматическая генерация тестов — важная вспомогательная часть верификации, ведь тесты, написанные вручную, не могут покрыть все возможные комбинации конструкций языка, а также все ситуации применения оптимизаций. QA команда компилятора Intel выбрала подход, использующий параметрическую контекстно-свободную грамматику, описываемую специальным мета-языком [1] для создания генератора тестов. Параметрические контекстно-свободные грамматики зарекомендовали себя в качестве хорошего формализма для построения генераторов семантически правильных и имеющих детерминированное поведение компиляторных тестов. Их отличала ясность описания генерируемых тестовых программ, а также гибкость и удобство в работе с их контекстом. Однако данный подход применялся только для генерации однопоточных тестов, исполняемых на центральном процессоре. Поэтому возник вопрос о возможности применения уже существующего генератора однопоточных тестов на основе параметрических контекстно-свободных грамматик, для генерации тестов для многопоточных расширений языка C++, а именно GFX-offload.

В данной работе представлен подход к автоматизации создания тестов для верификации GFX-offload компилятора, основанный на генераторе, использующем грамматики для порождения синтаксически корректных исполняемых тестов. Также приведены результаты использования полученной грамматики в процессе тестирования компилятора в компании Intel.

1. Проблема тестирования компиляторов. Язык программирования, строки которого являются входными данными для компилятора, определяется синтаксисом, статической семантикой и динамической семантикой. Основой для генерации тестов может стать любой из этих аспектов языка. Данные спецификации задают систему вложенных подмножеств всех возможных тестов.

Синтаксис языка определяется формальной грамматикой [2]. Грамматика состоит из следующих элементов:

— конечное множество нетерминальных символов;

— конечное множество терминальных символов;

— конечное множество продукционных правил;

— стартовый символ.

Цепочки терминалов, выводимые из стартового символа грамматики, называются синтаксически корректными программами. Увы, данные программы могут быть использованы только для проверки правильности работы синтаксического анализатора компилятора, так как они не всегда могут быть скомпилированы компилятором.

Статическая семантика, определяемая только для синтаксически корректных последовательностей, задает правила вычисления свойств программы, не требующих для этого ее выполнения. К таким свойствам относятся, например, типы переменных и выражений. Помимо правил вычисления, задаются также правила проверки статической корректности

программы — контекстные условия, накладывающие ограничения на возможные комбинации значений статических свойств программы. Программы, удовлетворяющие контекстным условиям, могут быть использованы для проверки работы статического анализатора компилятора.

Динамическая семантика определяет смысл выполнения статически корректных программ, то есть по сути это контекстные ограничения, которые должны быть выполнены во время исполнения программы. Для ее тестирования используются статически корректные программы, которые компилируются тестовым компилятором, исполняются, а затем результат их выполнения сравнивается с эталонным, определяемым динамической семантикой языка. Логично, что если тестовая программа не является детерминированной, то такое сравнение становится очень сложной задачей. В этой статье мы сделаем допущение, называя программу детерминированной, если она является динамически корректной. Конечно, это неправда в общем случае, но это приемлемо для целей тестирования.

В общем случае, для используемых на практике языков программирования, в частности C/C++, автоматическое выяснение детерминированности произвольной программы является очень сложной и, более того, алгоритмически неразрешимой задачей при статическом анализе [6].

Генерация синтаксически корректных тестов — не самая сложная задача и обычно успешно решается при помощи контекстно-свободной грамматики, которая используется для спецификации целевого языка. Генерация компилируемых (корректных с точки зрения статической семантики) программ — более сложная задача, потому что требует соответствия всем контекстным ограничениям целевого языка, которые обычно производятся при помощи контекстно-зависимых грамматик. Генерация детерминированных программ (корректных с точки зрения динамической семантики) является гораздо более сложной задачей.

Как правило, генераторы детерминированных программ представляют собой монолитные программы, написанные на высокоуровневом языке программирования. Обычно такие генераторы тяжело расширяемы и могут быть использованы только для генерации определенного класса программ. Также существует способ генерации, основанный на использовании некого набора предустановленных паттернов, с определенным набором вариаций, но этот подход также не гибок, и его сложность близка к ручному написанию тестовых программ.

Чтобы избежать написания еще одного монолитного тестового генератора, QA команда компилятора Intel выбрала подход, использующий параметрическую контекстно-свободную грамматику, описываемую специальным мета-языком [1]. Параметрические контекстно-свободные грамматики зарекомендовали себя в качестве хорошего формализма для построения генераторов семантически правильных и имеющих детерминированное поведение компиляторных тестов. Их отличала ясность описания генерируемых тестовых программ, а также гибкость и удобство в работе с их контекстом. Однако данный подход применялся только для генерации однопоточных тестов, исполняемых на центральном процессоре. Также оставался открытым вопрос — можно ли использовать его для эффективной генерации других классов тестов или есть более подходящие решения? В данной статье мы попытались дать ответ на этот вопрос.

2. Похожие работы. На данный момент существует множество работ, посвященных автоматической генерации тестов, поэтому мы опишем только основные.

В своих исследованиях К. В. Ханфорд [3] и П. Пардом [4] представили методики, позволяющие генерировать синтаксически корректные программы на основе их описания, без учета их семантических особенностей и контекстных ограничений.

В 1970 году К. В. Ханфорд [3] опубликовал работу, где предлагал способ генерации тестовых данных для компилятора PL/1 на основе динамической грамматики. В результате работы предложенного решения были получены синтаксически корректные программы, среди которых присутствовали семантически некорректные.

В свою очередь, алгоритм П. Пардома [4] использует контекстно-свободную грамматику, которая использует каждое продукционное правило не менее одного раза. Данный алгоритм не позволял учитывать контекстные правила целевого языка программирования и мог использоваться только для генерации тестов для парсеров и синтаксических анализаторов компилятора. Однако данная работа стала фундаментальной для всех работ, связанных с генерацией тестов для компиляторов, и послужила отправной точкой для последующих исследований.

В работе 1981 года А. Г. Данкэн и Дж. С. Хатчисон [5] предложили использовать расширение контекстно-свободных грамматик — атрибутные грамматики — для описания входных данных для тестового генератора. Во время генерации последовательно раскрываются все нетерминальные символы, если это позволяют сделать контекстные ограничения. В результате получается набор синтаксически и семантически корректных тестов, покрывающих все продукционные правила грамматики и все контекстные условия. Это позволяет только вести анализ выполненных контекстных условий. Но это ведет к большому числу холостых прогонов генератора, когда из-за невыполненных контекстных условий приходится останавливать процесс генерации, а также к большому числу семантически неинтересных тестов.

Базичи и Спадафора [6] представили новый способ, в котором генератор опирается на специально формализованное описание исходного языка. Этот формализм расширял контекстно-свободные грамматики в сторону контекстно-зависимых, но сохранял структуру и удобство BNF нотации. К сожалению, авторы предлагали использовать нетерминалы для распознавания левых частей правил, что могло занимать много времени в случае длинных цепочек символов. Также статья не решала проблемы по нахождению кратчайших правил грамматики.

Работу А. П. Статсенко [1] можно рассматривать в качестве продолжения трудов Базичи и Спадафоры. Она использует параметрические контекстно-свободные грамматики для генерации синтаксически и семантически корректных тестовых программ для C/C++/Fortran компиляторов. Так как мы выбрали использование данного формализма, он будет разъяснен позже.

3. Описание подхода. В рамках данной работы мы создали на основе существующей параметрической контекстно-свободной грамматики такую, которая позволила генерировать тесты для проверки расширения языка C++ GFX-offload, позволяющего производить гетерогенные вычисления на связке CPU, и интегрированной в большинство консьюмерских процессоров — графики. Мы сфокусировались на генерации „параллельных“ версий обычных C циклов, т. к. в будущем можно легко воспользоваться этими наработками для генерации тестов для других расширений языка.

Фокус на генерации различных параллельных циклов позволяет улучшить валидацию векторизующих оптимизаций компилятора, которые играют важную роль в улучшении

производительности на современных вычислительных архитектурах. GFX-offload расширение имеет следующие ограничения для циклов:

— запрет на переходы из цикла (return, break или goto);

— ограничения на переменную цикла;

— ограничения на смешивание конструкций различных параллелизационных расширений (например, offload блок не должен содержать Cilkspawn или Cilksync).

4. Краткое формальное описание параметрической контекстно-свободной грамматики и генератора. Для описания грамматики используется специализированный язык, напоминающий BNF-нотацию и функциональный язык. Грамматика задается в одном файле и является набором строк с поддержкой комментариев. Правило — это идентификатор, для которого задано, каким образом он может быть переписан. Правило может быть как генерирующим, так и ограничивающим. Любой идентификатор — это последовательность символов, не начинающаяся с цифры. Имя параметра шаблона контекста в левой части правила может быть любым именем распознанной группы символов регулярного выражения языка Python, стоящей в предыдущем шаблоне контекста левой части правила. Правая часть правила может задаваться либо альтернативами, либо многострочным правилом.

Для более тонкой работы с правилами могут быть использованы специальные распознаватели контекста, при этом число таких распознавателей ограничено только здравым смыслом. Основное их удобство состоит в возможности не только извлечения объектов из контекста и их переименования, но и возможности изменить сам контекст для всех идентификаторов в правой части правила.

Помимо описанного механизма распознавателей, существует еще один способ описать и распознать контекст — условия. Условие — это выражение, вычисляющееся в булевский тип. А если условий несколько, то они объединяются по принципу логического „И“. Также в языке грамматики предусмотрено несколько встроенных функций для удобной работы со списками, условиями и числами.

Данная параметрическая контекстно-свободная грамматика по наглядности сопоставима с обычной контекстно-свободной грамматикой, но позволяет генерировать более широкий класс контекстно-зависимых языков.

При генерации выбор правил осуществляется случайно, что не гарантирует их достижимости, однако, как показывают другие исследования в этой области, такая проблема характерна и для других подходов к тестированию компиляторов [7].

Используемый в данной работе генератор подробно описан в статье Стасенко А. П. „Генерация исполняемых тестов для компилятора“ [1].

Одного генератора и грамматики, порождающей тестовые программы, мало: для использования в процессе тестирования нужна система, которая будет валидировать сгенерированные тесты, как в целях исследования их полезности для компилятора, так и в целях исключения тестовых проблем из-за потенциальной недетерменированности полученной программы. И такая система была создана авторами генератора на языке Perl, названа AutomaticTestGenerator (в дальнейшем — ATG) и позволяла в автоматическом режиме тестировать компилятор при помощи тестов, порождаемых тестовым генератором. Схема данной системы изображена на рисунке.

Давайте подробнее рассмотрим процесс работы всей системы. Каждый сгенерированный тест проходит 2 проверки: компиляция с оптимизациями и без. То есть, если тест успешно компилировался при помощи компилятора с отключенными оптимизациями и

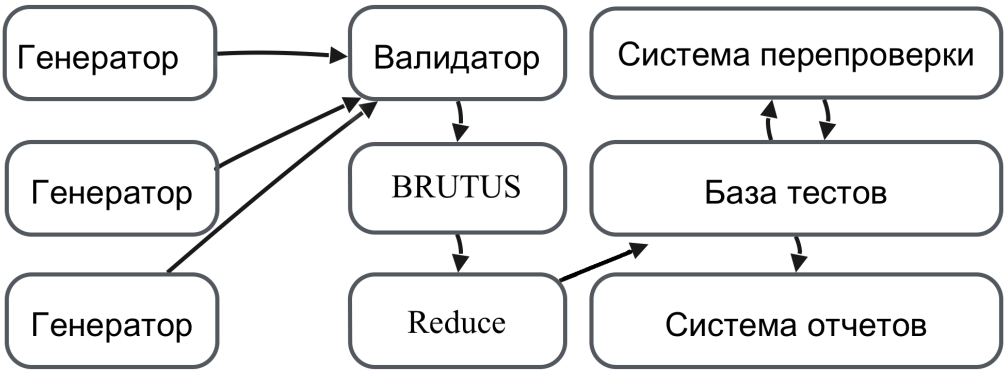

Рис. 1. Схема системы ATG

включенной проверкой указателей и запускался, значит, он является статически и динамически детерминированной программой, и результат ее исполнения является эталонным. Здесь стоит сделать уточнение насчет успешного запуска: успешный запуск — это отсутствие сообщений о проблемах с указателями и нулевой код завершения, а также отсутствие завершения программы по установленному тайм-ауту. Это первая ступень проверки. Следующая ступень — это проверка компиляции с оптимизациями. Если тест вызвал на данном этапе проблемы, значит он уже полезен для тестирования и сохраняется для дальнейшей обработки. Если компиляция прошла успешно, то проверяется корректность исполнения программы с ее эталоном. И если программа проваливает эту проверку, то она также сохраняется. В случае, когда программа проходит проверку с эталоном, она просто удаляется, и запускается процесс генерации нового теста.

Сохраненные тесты проходят дальнейшую обработку при помощи инструмента редукции программного кода Reduce для выделения той части теста, которая и приводит к проблемам в компиляторе. Также производится попытка найти компоненту компилятора, виновную в проблеме при помощи функционала компилятора, который позволяет отключать применяемые оптимизации. Затем тесты сохраняются в базу системы с пометкой, на какой системе, когда и с какой проблемой они были получены.

С целью оптимизации места, занимаемого хранилищем тестов, сохраняются 10 минимальных по размеру тестов на каждую проблемную оптимизацию. Но если возможно найти дату проблемной сборки, то это позволяет сохранять по 10 тестов на каждую дату. Система периодически производит перепроверку всех тестов из базы, помечая, на каких платформах тесты проходят, а на каких „падают“. Помимо проверки на последней версии компилятора, производится проверка и на продуктовых бранчах компилятора, чтобы не допустить попадания таких ошибок в конечный продукт, а еще, чтобы выполнить требования процесса тестирования компилятора. Система раз в сутки высылает отчет аналитику о найденных проблемах. В дальнейшем аналитик создает дефект на компилятор и прикладывает к нему тест для воспроизведения проблемы, помечая в заголовке теста, что для него был создан дефект с определенным идентификатором. Состояние тестов, помеченных дефектом, также отслеживается в ежедневном отчете, а именно: в случае, если дефект был закрыт, а тест продолжает падать, значит, аналитику нужно обратить на него особое внимание.

5. Создание грамматики для GFX-offload компилятора. В качестве основы для грамматики была взята уже существующая грамматика для генерации тестовых программ

на языке C++. Из нее были взяты некоторые стандартные правила (генерация циклов, условий и т. д.).

В первую очередь хотелось проверить качество кодогенератора компилятора для новой платформы. Поэтому было принято решение перенести исполнение большей части теста на плечи интегрированного графического процессора. Как это реализовано в грамматике для генератора, показано в примере. Однако исполнение offload программы имеет некоторые ограничения, например, такие как запрет на вызов не помеченных как offload функций, запрет на операции со стандартным вводом-выводом и несколько других. Поэтому все несовместимые синтаксические конструкции были вынесены за пределы offload части программы.

Пример: правила, добавляющие offload прагму в тест

```
offloadpragmactx locals globals ::=
{"_Pragma (\"offload target(gfx) inout("
    print-arrays(ctx) print-gpointers(ctx) ")\")"
"_Pragma (\"parallel_loop\")"

"for (int i_i = 0; i_i < 1; i_i++){"
        declarations(locals, globals)
        statements(ctx)
        for-clause(ctx)
        statements(ctx)
"}"
}
print-arrays (lv:rv:as:_) ::= arr-list(as)
arr-list () ::= ""
arr-listqw:() ::= " " get(qw,0)
arr-list qw:qws ::= " " get(qw,0) "," arr-list(qws)
print-gpointers (lv:_) ::= glob-list(lv)
glob-list () ::= ""
glob-listqw:() ? is-prefix(qw, "g_")::= ", " qw
glob-listqw:qws ? is-prefix(qw, "g_")::= ", " qw glob-list(qws)
glob-list qw:() ::= ""
glob-listqw:qws::= glob-list(qws)
```

Как можно заметить, для корректности работы с глобальными и локальными для функции main переменными они добавляются внутрь offload цикла при помощи опции offload блока inout.

Все глобальные переменные, а также вызываемые функции помечаются в соответствующих правилах при помощи добавления __declspec(target(gfx)) для возможности вызова и использования оных внутри offload части теста. Самым простым offload блоком является цикл for, поэтому основной блок программы был помещен в цикл с одной итерацией, помеченный offload прагмой, и указанием всех переменных, участвующих в вычислениях на графическом процессоре. После окончания этой offload части, производится вывод всех глобальных переменных и массивов в стандартный поток вывода. Это необходимо для обнаружения ошибок времени исполнения.

Так как, помимо проверки работы оптимизирующих алгоритмов компилятора, мы хотели проверить работу кодогенератора, валидация тестов должна представлять собой следующую процедуру: компиляция тестовой программы с отключенной offload прагмой (чтобы тест исполнялся на центральном процессоре) и ее запуск считался бы эталонным запуском, а компиляция offload-компилятором с максимальным уровнем оптимизаций и запуск программы с использованием графического ядра считался бы тестовым, и происходило бы сравнение результатов этих запусков. К сожалению, в первое время данный подход не сработал из-за найденной в компиляторе проблемы, и пришлось временно перейти на процедуру валидации, в которой эталонный результат получался компиляцией offload-компилятором с отключенными оптимизациями и запуском скомпилированной программы на графическом процессоре. Конечно, в этом случае мы потеряли возможность находить целый класс ошибок, но в то же время это позволило продолжать получать новые тесты и продолжать находить новые проблемы в компиляторе.

Особых проблем с динамической корректностью генерируемых тестов не было из-за выбранного подхода по генерации одного всеобъемлющего цикла с одной итерацией, без параллелизма.

Возникли определенные трудности и при работе Reduce, который минимизирует исходный код тестов. Как можно заметить, вместо директив #pragma, используются _Pragma, это было сделано, чтобы разделить обычные директивы от добавленных в грамматику. Такое разделение необходимо, так как ретранслятор Reduce при восстановлении исходного кода из синтаксического дерева объединял лексемы директив и цикла, и в результате при попытке удаления вершины с директивой offload возникала синтаксическая ошибка. Поэтому все особые директивы добавляются при помощи _Pragma, удаление которых запрещено в правилах минимизации Reduce.

Заключение. Апробация и внедрение результатов исследования были произведены в компании Intel для тестирования их компилятора. После всех необходимых изменений в AutomaticTestGenerator, он был запущен с этими грамматиками, и первые результаты дал генератор GFX-offload тестов. В бета версии компилятора было найдено 8 ошибок, по большей части на стадии компиляции, но две из них — во время исполнения. Из этих ошибок только одна находилась не в компоненте компилятора, отвечающей за векторизацию, а во внутреннем представлении.

Мы убедились, что формализм параметрических контекстно-свободных грамматик оказался удобным инструментом для генерации детерменированных параллельных тестовых программ. Хотя, конечно, еще необходимо внести некоторые изменения в соответствующие грамматики или поменять глубину рекурсии генератора, чтобы усложнить тесты, а значит сделать генераторы более продуктивными.

Список литературы

1. Стасенко А. П. Генерация исполняемых тестов для компилятора // Конструирование и оптимизация параллельных программ. Серия „Конструирование и оптимизация программ“. Новосибирск, 2008. С. 301–313.

2. Aho A. V., Sethi R., Ullman J. D. Compilers: principles, techniques, and tools. Boston: Addison-Wesley Longman Publishing Co., Inc., 1986. P. 796.

3. Hanford K. V. Automatic generation of test cases // IBM Systems Journal. NY: IBM, Dec. 1970. Vol. 9. P. 242–257.

4. Purdom P. A sentence generator for testing parsers // Behavior and Information Technology, July 1972. Vol. 12. N 3. P. 366–375.

5. Hutchison J. S., Duncan A. G. Using Attributed Grammars to Test Designs and Implementation // In Proc. of the 5th international conference on Software engineering, 1981. P. 170–178.

6. Bazzichi F, Spadafora I. An automatic generator for compiler testing // IEEE transactions on Software Engineering. NY: IEEE, 1982. Vol. SE-8. P. 343–353.

7. Kossatchev A. S., Posypkin M. A. Survey of compiler testing methods // Programming and Computing Software. NY: Plenum Press, 2005. Vol. 31, N 1. P. 10–19.

Панкратов Святослав Борисович — аспирант института систем информатики им. А. П. Ершова СО РАН; e-mail: aquaxpi@gmail.com; тел.: +7 (913) 482-0939,

Святослав Панкратов окончил бакалавриат физического факультета Новосибирского государственного университета на кафедре автоматизации физико-технических исследований в 2012 году и в качестве дипломной работы создал систему редукции программ на языке Fortran. В 2014 году закончил магистратуру на кафедре АФТИ с дипломной работой, посвященной автоматической генерации тестов для компилятора. В 2014-м году поступил в аспирантуру Института систем информатики СО РАН и продолжил работу над автоматическим тестовым генератором. С 2010 года является сотрудником компании Intel, на базе которой выполнялись исследования для научных работ в сфере тестирования компиляторов. Его текущие исследовательские интересы включают технологии параллельных вычислений, компиляторные оптимизации, автоматизацию тестирования, облачные языки программирования и машинное обучение. В данное время основным проектом С. Панкратова является быстро развертываемая система автоматической генерации тестов для компиляторов, делающая акцент на тестировании параллельных расширений и оптимизаций.

Pankratov Svyatoslav received his B. S. degree in Physics from the Novosibirsk State University at 2012 with diploma work about source code reducer for Fortran programming language. He continued his study and received M.S. degree with work about automatic tests generation for optimizing compiler at 2014. As next postgraduate step he started (2014) a Ph.D. program in Computer Science in Institute of Informatics System of the Russian Academy of Sciences (RAS), Siberian Branch. From 2010 he had a job at Intel Corporation. Intel is a base for his researches in the compiler's testing field. His current research interests include parallel computing technologies, compiler's optimizations, testing automation, cloud programming languages and machine learning. The main project now is rapid-deployable automated test generator for compilers, with emphasis on testing parallel extensions and optimizations.

Дата поступления — 25.05.2017