Верификация компиляторов – систематический подход[1]

С.В. Зеленов, Н.В. Пакулин

Аннотация. Компиляторы преобразуют исходный текст программы на языке высокого уровня в вид, пригодный для исполнения. Ошибки в компиляторе чреваты отказами или ошибками при выполнении исполняемых файлов, построенных компилятором, поэтому обеспечение корректности функционирования компилятора является важнейшей задачей. Одна из особенностей компиляторов заключается в том, что они принимают на вход и выдают данные с очень сложной структурой, поэтому при проведении верификации на практике можно исследовать поведение компилятора лишь на небольшом подмножестве входных программ. В статье представлен подход к верификации компиляторов, основанный на декомпозиции общей задачи компилятора, и представлены методы решения выделенных задач. Представленный подход использовался при верификации различных индустриальных компиляторов и трансляторов.

1. Введение

Языки высокого уровня являются основным средством разработки программных систем. Спецификация языка высокого уровня задаёт класс текстов, принадлежащих этому языку, и определяет семантику исполнения программ, написанных на этом языке. Задача перевода текстов с языка высокого уровня в представление, выполнимое на вычислительной системе, решается комплексами программ, которые по традиции называют компиляторами.

Ошибки в компиляторах приводят к тому, что выполнение результирующих исполнимых модулей отличается от поведения, определяемого спецификацией языка. Дефекты исполнимых модулей, вызванные ошибками в компиляторе, очень сложно выявлять и исправлять, и в целом корректность исполнимых модулей, построенных некорректным компилятором, вызывает сомнения. Корректность компилятора является необходимым требованием корректной и надёжной работы программного обеспечения, разработанного на соответствующем языке высокого уровня. Тем самым, задачу верификации[2]

[1] Работа частично поддержана грантом РФФИ 05-01-00999.

[2] Согласно Модели зрелости процесса разработки (Capability Maturity Model Integration) «верификация» определяется следующим образом: Верификация

компилятора можно считать одним из важнейших средств обеспечения надёжности программного обеспечения.

Основной источник трудностей при верификации компилятора заключается в том, что компилятор принимает на вход данные со сложной структурой, большим количеством внутренних связей и обладающие очень сложной семантикой времени исполнения. Для снижения сложности задачи верификации компилятора мы предлагаем декомпозировать её на отдельные подзадачи, которые в совокупности покрывают всю функциональность компилятора[3].

Функциональность, типичную для большинства компиляторов, можно условно декомпозировать на следующие задачи:

1. Анализ синтаксической корректности исходного теста программы.
2. Анализ выполнения контекстных условий в исходном тексте[4].
3. Оптимизация внутреннего представления и генерация выходных данных.

Результатом работы компилятора является объектный модуль на машинном языке или модуль на исполнимом промежуточном языке, таком как байт-код Java и Python или Intermediate Language платформы .NET.

Для большинства компиляторов языков программирования, используемых на практике, генерируемый код исполняется в специализированном окружении, которое обобщённо называется run time support. В рамках данной статьи мы будем пользоваться термином «подсистема поддержки исполнения».

Подсистема поддержки исполнения тесно связана с компилятором. Например, каждый компилятор Си и Си++ поставляется со своим набором стандартных библиотек, причём библиотеки одного компилятора нельзя использовать для построения исполнимых файлов другим компилятором. Даже в тех случаях, когда промежуточный язык и стандартные библиотеки стандартизированы, существуют тесные зависимости между компилятором и средой исполнения. Например, для исполнения байт-кода Java в виртуальной машине Java компании IBM лучше всего для компиляции использовать компилятор Java именно этой фирмы; аналогично для исполнения байт-кода в виртуальной

обеспечивает соответствие набора результатов работы требованиям, которые для них заданы [1].

[3] Отметим, однако, что предлагаемый подход не отменяет интеграционного тестирования компилятора.

[4] Примеры контекстных условий: требование, чтобы все используемые переменные, методы и т.д. были объявлены в коде программы; требование, чтобы аргументы вызова метода по своему количеству и по своим типам соответствовали формальным параметрам вызываемого метода.

машине компании Sun следует компилировать исходные тексты компилятором Java, разработанным в Sun.

В настоящей работе представлен подход к верификации компиляторов, основанный на технологии автоматизированного тестирования UniTESK[4]. Методы, составляющие представленный подход, построены по единому шаблону формализации требований [9].

2. Систематический подход к верификации функций компилятора

Функции компилятора тесно связаны, поэтому при верификации компилятора необходимо проверять корректность каждой из них. В предлагаемом подходе задача верификации компилятора декомпозируется на основе декомпозиции функциональности компилятора. Мы выделяем следующие задачи:

1. Верификация синтаксического анализатора компилятора.
2. Верификация семантического анализатора компилятора.
3. Верификация оптимизаций и генерации кода.
4. Верификация библиотек поддержки времени исполнения (runtime).

Некорректное функционирование подсистемы поддержки исполнения делает почти бесполезным весь компилятор – результат компиляции будет работать неверно из-за ошибок в этой подсистеме и, следовательно, станет непригодным для нужд пользователя. Тесная связь компилятора и подсистемы поддержки исполнения служит обоснованием того, чтобы включить верификацию подсистемы поддержки исполнения в общую задачу верификации компилятора.

Корректность оптимизаций и генерации кода определяет корректность создаваемых транслятором выходных данных. Поэтому верификация этой функциональности является основной задачей верификации компилятора.

От корректности синтаксического анализа и анализа контекстных условий зависит корректность основной функциональности транслятора – оптимизирующих преобразований, генерации кода. Поэтому решение задачи верификации анализа синтаксиса и контекстных условий является базой для решения задач верификации остальных функций компилятора.

В литературе выделяют два вида верификации – статическая верификация средствами анализа кода и аналитической верификации и динамическая верификация посредством тестирования. Компилятор любого практически полезного языка представляет собой настолько сложную систему, что статической верификации поддаются только отдельные небольшие подсистемы, например, отдельные блоки оптимизации. Для решения задач верификации, поставленных выше, на практике используются методы динамической верификации.

В связи с этим при проведении верификации компилятора и его функций необходимо решить следующий набор задач:

1. автоматизация построения тестов:
   a. автоматизация генерации тестовых данных;
   b. автоматизация проверки корректности обработки тестовых данных (проблема построения оракула);
2. определение критерия завершения верификации.

В данной работе мы предлагаем решать эти задачи на систематической основе. Во всех задачах верификации методы верификации строятся по единой схеме (рис. 1).

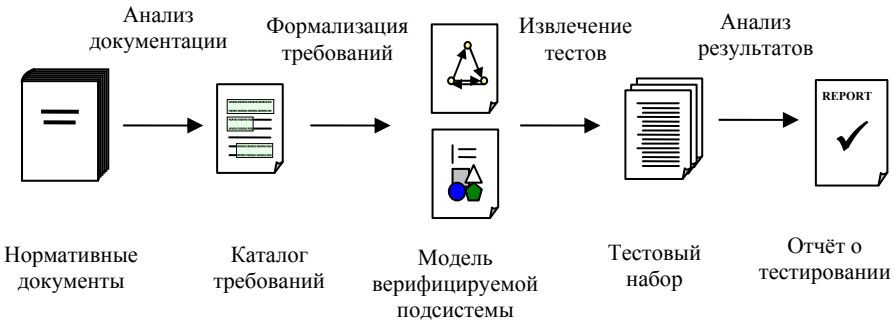

Рис. 1. Общая схема метода верификации

Эта схема метода верификации состоит из нескольких этапов. На первом этапе требования извлекаются из нормативных документов и систематизируются. В результате получается *каталог требований*, в котором требования сформулированы максимально однозначно, требования классифицированы, и, возможно, установлены связи между отдельными требованиями. Каталог требований используется на последующих этапах.

Второй этап нацелен на представление требований в формальном виде. Требования из каталога записываются с использованием того или иного математического формализма. Такая запись требований называется формальной спецификаций или формальной моделью.

На третьем этапе на основе построенной модели автоматизировано генерируются тесты. В зависимости от задачи тесты могут быть либо просто тестовыми данными, либо дополнительно содержать оракул для автоматического вынесения вердикта о корректности наблюдаемого поведения компилятора.

В результате исполнения тестов строятся отчёты о тестировании. В них содержится информация о том, насколько наблюдаемое поведение

компилятора соответствует формальной модели. Вопросы анализа вердиктов, обнаружения дефектов и их исправления выходят за рамки данной статьи.

3. Верификация синтаксического анализатора

Синтаксический анализ входного текста является частью функциональности любого транслятора. *Парсером* мы называем булевскую функцию, заданную на множестве последовательностей токенов и принимающую значение «истина», если последовательность является предложением данного формального языка, и «ложь» – в противном случае.

Задачу верификации парсера можно разделить на следующие подзадачи:

1. Проверка того, что парсер принимает предложения, принадлежащие целевому языку.
2. Проверка того, что парсер отвергает предложения, не принадлежащие целевому языку.

3.1. Критерии полноты тестирования

Мы рассматриваем известные алгоритмы синтаксического анализа (LL-анализ и LR-анализ, построенные на основе стека — см. [2]) в качестве алгоритмов, моделирующих поведение парсера.

Критерий полноты набора позитивных[5] тестов для тестирования LL-анализатора:

(PLL) Покрытие всех пар вида

(нетерминал A, допустимый следующий токен t),

где пара (`A, t`) считается покрытой тогда и только тогда, когда в тестовом наборе существует предложение языка, обрабатывая которое LL-анализатор придет в ситуацию, когда на вершине стека будет находиться символ `A`, а текущим входным символом будет токен `t`.

Критерий полноты набора позитивных тестов для тестирования LR-анализатора:

(PLR) Покрытие всех пар вида

(символ s состояния конечного автомата,
помеченный символом X переход из состояния s),

где пара (s, X) считается покрытой тогда и только тогда, когда в тестовом наборе существует предложение языка, обрабатывая которое LR-анализатор придет в ситуацию, когда на вершине стека будет находиться символ s, а началом текущего входного потока будет последовательность токенов, отвечающая символу X.

Критерий полноты набора негативных[6] тестов для тестирования LL-анализатора:

(NLL_R) Покрытие всех пар

(нетерминал A; «некорректный» токен t'),

где пара (A, t') считается покрытой тогда и только тогда, когда среди тестов имеется последовательность токенов, не принадлежащая целевому языку, такая, что LL-анализатор, обрабатывая эту последовательность, придет в ситуацию, когда после обработки как минимум R «правильных» токенов, на вершине стека будет находиться символ A, а текущим входным символом будет «некорректный» токен t'.

Критерий полноты набора негативных тестов для тестирования LR-анализатора:

(NLR_R) Покрытие всех пар

(символ s состояния конечного автомата; «некорректный» токен t'),

где пара (s, t') считается покрытой тогда и только тогда, когда среди тестов имеется последовательность токенов, не принадлежащая целевому языку, такая, что LR-анализатор, обрабатывая эту последовательность, придет в ситуацию, когда после обработки как минимум R «правильных» токенов, на вершине стека будет находиться символ s, а текущим входным символом будет «некорректный» токен t'.

Существует семейство алгоритмов генерации тестов, удовлетворяющих этим критериям. Их подробное описание можно найти в работе [6].

Следует особо отметить следующее важное свойство предложенных в этой работе алгоритмов генерации негативных тестов: алгоритмы гарантированно строят тесты, не принадлежащие целевому формальному языку. Это свойство позволяет отказаться при построении оракула от использования каких бы то ни было эталонных синтаксических анализаторов.

Разработан инструмент SynTesK [14], в котором реализованы алгоритмы, удовлетворяющие критериям PLL и NLL_1, а также критерию NLR_1 для некоторого специального вида грамматик [6].

Инструмент SynTesK успешно применялся для тестирования компиляторов реальных языков программирования, в т.ч. C и Java [6]. Практические

[5] Позитивный тест для парсера – это последовательность токенов, на которой парсер выдает вердикт «истина», т.е. последовательность токенов, являющаяся предложением целевого языка.

[6] Негативный тест для парсера – это последовательность токенов, на которой парсер выдает вердикт «ложь», т.е. последовательность токенов, не являющаяся предложением целевого языка.

результаты этого применения показали эффективность реализованных в SynTesK алгоритмов.

3.2. Регламентирующие документы. Анализ требований

В рамках предлагаемого подхода под тестированием парсера понимается проверка соответствия парсера функциональным требованиям.

Функциональные требования извлекаются, как правило, из описания стандарта языка. Основное функциональное требование для парсера определяется множеством корректных предложений целевого языка. Обычно это множество описывается при помощи контекстно-свободной грамматики. Часто в тексте стандарта языка уже присутствует соответствующее описание в формальном виде.

3.3. Моделирование

Для формального задания модели целевого языка используется представление грамматики языка в форме BNF.

3.4. Извлечение тестов

Инструмент SynTesK предоставляет возможность в полностью автоматическом режиме систематически генерировать наборы тестов, удовлетворяющих сформулированным выше критериям покрытия.

В качестве входных данных генератор тестов принимает формальное описание грамматики целевого языка в форме BNF.

3.5. Тестирование

Автоматический оракул для прогона позитивных тестов устроен так: проверяется, что результатом обработки каждого теста парсером является значение «истина». При применении тестов к реальному компилятору значение парсера «истина» трактуется следующим образом – в выводе компилятора отсутствуют сообщения о синтаксических ошибках.

Аналогично устроен оракул для прогона негативных тестов: проверяется, что результатом обработки каждого теста парсером является значение «ложь», то есть в выводе компилятора присутствуют сообщения об ошибках синтаксиса.

4. Тестирование анализаторов контекстных условий

Анализатором контекстных условий мы называем булевскую функцию, заданную на множестве предложений данного формального языка[7] и принимающую значение «истина», если в предложении выполняются все относящиеся к нему контекстные условия, и «ложь» – если хотя бы одно из этих контекстных условий нарушается.

[7] Имеются в виду все синтаксически корректные предложения.

4.1. Методы тестирования

Метод SemaTesK [3] автоматической генерации тестов для тестирования анализаторов контекстных условий основан на использовании модели атрибутированных деревьев для конструктивного описания статической семантики целевого языка. Метод позволяет получать наборы как позитивных[8], так и негативных[9] тестов для тестирования анализатора контекстных условий.

В методе SemaTesK используется формальное описание контекстных условий в форме зависимостей между вершинами абстрактного синтаксического дерева. Зависимость между некоторыми двумя вершинами индуцируется зависимостями между соответствующими атрибутами этих вершин. Одна из вершин, участвующих в контекстном условии, называется *источником*, а другая — *целью* в том смысле, что атрибуты вершины-цели зависят от атрибутов вершины-источника.

Этот метод дает возможность локализовать описание одного неформального контекстного условия в пределах одного формального правила, а также позволяет в подавляющем большинстве случаев задавать контекстные условия декларативно[10].

Построение тестов методом SemaTesK производится в соответствии со следующими критериями покрытия расположения контекстных условий в различных синтаксических контекстах.

Критерий полноты набора позитивных тестов для тестирования анализатора контекстных условий:

Для каждого контекстного условия R из множества всех контекстных условий набор тестов должен содержать тексты, покрывающие R и такие, что соответствующее множество деревьев содержит все возможные вершины «источник» и «цель» во всех возможных синтаксических контекстах.

Критерий полноты набора негативных тестов для тестирования анализатора контекстных условий:

[8] Позитивный тест для анализатора контекстных условий – это такое синтаксически корректное предложение, на котором анализатор контекстных условий выдает вердикт «истина», т.е. предложение, корректное с точки зрения статической семантики целевого языка.

[9] Негативный тест для анализатора контекстных условий – это такое синтаксически корректное предложение, на котором анализатор контекстных условий выдает вердикт «ложь», т.е. предложение, не корректное с точки зрения статической семантики целевого языка.

[10] В частности, данным методом была описана статическая семантика языков C и Java.

Для каждой мутации R^- каждого контекстного условия R из множества всех контекстных условий набор тестов должен содержать множество текстов, покрывающих R^- и таких, что соответствующее множество деревьев содержит все возможные вершины «источник» и «цель» во всех возможных синтаксических контекстах.

Более подробное описание метода SemaTesK можно найти в работе [3].

Следует особо отметить следующее важное свойство метода SemaTesK. Метод дает возможность непосредственно строить тесты, нацеленные на проверку корректности анализа контекстных условий. Это свойство позволяет в отличие от прочих известных методов отказаться при генерации тестов от использования фильтрации (селекции) позитивных/негативных тестов среди множества всех синтаксически корректных предложений данного языка, а также позволяет систематически генерировать тесты, удовлетворяющие адекватным критериям полноты.

Метод SemaTesK успешно применялся для тестирования компиляторов реальных языков программирования, в т.ч. C и Java [3]. Практические результаты этого применения показали эффективность метода SemaTesK.

Заметим, что особенно перспективным направлением использования метода SemaTesK является генерация тестов для разрабатываемых диалектов языков, так как вносить небольшие изменения в их описания достаточно просто, и таким образом количество ручного труда для приведения тестов в соответствие с очередной редакцией языка не велико.

4.2. Регламентирующие документы. Анализ требований

В рамках метода SemaTesK под тестированием анализатора контекстных условий понимается проверка соответствия анализатора контекстных условий функциональным требованиям.

Функциональные требования извлекаются, как правило, из описания стандарта языка. Основное функциональное требование для анализатора контекстных условий определяется множеством предложений целевого языка, корректных с точки зрения статической семантики. Обычно это множество задается в стандарте при помощи описания синтаксиса языка в виде контекстно-свободной грамматики, а также при помощи описания контекстных условий в виде неформального текста. Основной работой по анализу требований в рамках метода SemaTesK является извлечение из стандарта языка и составление списка всех контекстных условий.

4.3. Моделирование

Модель синтаксиса целевого языка задается в виде формального описания на языке TreeDL [5] структуры корректных абстрактных синтаксических деревьев целевого языка.

Модель статической семантики целевого языка задается в виде списка контекстных условий. Для их формального описания используется язык SRL (Semantic Relation Language), специально разработанный для этой цели [3].

4.4. Извлечение тестов

Имеется инструментальная поддержка метода SemaTesK, которая позволяет в полностью автоматическом режиме систематически генерировать наборы тестов, удовлетворяющих сформулированным выше критериям покрытия.

В качестве входных данных генератор тестов принимает формальное описание структуры абстрактных синтаксических деревьев целевого языка на языке TreeDL, а также описание контекстных условий целевого языка на языке SRL.

4.5. Тестирование

Автоматический оракул для прогона позитивных тестов устроен так: проверяется, что результатом обработки каждого теста анализатором контекстных условий является значение «истина». При применении тестов к реальному компилятору значение парсера «истина» трактуется следующим образом – в выводе компилятора отсутствуют сообщения о семантических ошибках.

Аналогично устроен оракул для прогона негативных тестов: проверяется, что результатом обработки каждого теста анализатором контекстных условий является значение «ложь», то есть в выводе компилятора присутствуют сообщения об ошибках семантики.

5. Тестирование оптимизаций и генерации кода

Back-end'ом мы называем функцию, которая принимает на вход структуру обрабатываемых входных данных в некотором внутреннем представлении и генерирует соответствующие ей требуемые выходные данные. Функциональность back-end'а касается следующих аспектов:

- анализ внутреннего представления программы, проводящийся с целью обогащения его дополнительной информацией, которая требуется при генерации выходных данных — примерами подобного анализа являются, например, анализ графа потока управления программы, анализ потоков данных в программе и проч.;
- трансформации (оптимизации) внутреннего представления программы, проводящиеся с целью улучшения качества выходных данных в соответствии с выбранным критерием (например, размер или скорость работы программы); примерами подобных оптимизаций являются, например, удаление неиспользуемых

частей входных данных, вычисление константных выражений на этапе трансляции, оптимизация размещения аргументов и промежуточных результатов вычислений в регистрах процессора и проч.;

- построение результирующего представления в выходном языке.

5.1. Методы тестирования

Метод OTK [7] автоматической генерации тестов для тестирования back-end'а основан на построении модели его входных данных на основе из абстрактного описания алгоритма работы back-end'а.

Метод OTK позволяет строить модели данных и разрабатывать на их основе генераторы тестов для компилятора, нацеленных на тестирование данного аспекта back-end'а.

Модель данных строится на основе абстрактного описания алгоритма работы тестируемого аспекта.

Алгоритм формулируется с использованием *терминов*, обозначающих сущности некоторого подходящего абстрактного представления входных данных, такого как граф потока управления, граф потока данных, таблица символов и пр. Back-end для осуществления своих действий ищет сочетания сущностей абстрактного представления программы, которые удовлетворяют некоторым *шаблонам* (например, наличие в программе циклов, наличие в теле цикла конструкций с определенными свойствами, наличие в процедуре общих подвыражений, наличие между инструкциями зависимости данных некоторого вида и пр.). При этом могут учитываться сущности лишь части терминов. Для построения модели данных рассматриваются только те термины, которые именуют сущности, задействованные хотя бы в одном шаблоне.

Итак, в результате анализа алгоритма выделяются термины и шаблоны, используемые в алгоритме. Далее на основании этой информации описывается множество модельных строительных блоков:

- каждому термину соответствует свой вид модельного строительного блока;
- строительные блоки могут связываться между собой и образовывать структуры, соответствующие шаблонам.

Модельной структурой называется граф, вершины которого – строительные блоки, а ребра – связи между строительными блоками. Модельная структура является абстракцией той части внутреннего представления структуры программы, которая существенна для целевого аспекта back-end'а.

Критерий полноты множества тестов формулируется в терминах построенной модели на основе информации о тех шаблонах, которые были выделены при анализе алгоритма.

Более подробное описание метода OTK можно найти в работе [7].

Метод OTK успешно применялся при тестировании оптимизирующих компиляторов GCC, Open64, Intel C/Fortran compiler (см. [7]), а также для тестирования оптимизаторов графических моделей [8], реализации протокола IPv6, подсистемы построения отчетов в биллинговой системе, обработчиков XML-логов [4]. Практические результаты этого применения показали эффективность метода OTK.

5.2. Регламентирующие документы. Анализ требований

В рамках метода OTK под тестированием back-end'а в трансляторе понимается проверка соответствия семантики выходных данных, созданных back-end'ом, семантике соответствующих входных данных.

Структура входных данных извлекается из абстрактного описания алгоритма работы back-end'а (см. выше описание метода тестирования).

5.3. Моделирование

Модель структуры входных данных задается в виде формального описания на языке TreeDL [5].

5.4. Извлечение тестов

Для извлечения тестов в рамках метода OTK требуется разработать генератор тестов.

Генератор тестов состоит из двух компонентов. Первый, называемый *итератором*, отвечает за последовательную генерацию модельных структур в соответствии с выбранным критерием полноты. Второй компонент, называемый *меппером*, отвечает за отображение каждой модельной структуры в соответствующую корректную программу на языке входных данных компилятора.

В рамках метода OTK предоставляется мощная инструментальная поддержка для формального описания модели данных, а также для разработки всех требующихся компонентов генератора тестов [13].

Генератор, разработанный с помощью OTK, позволяет для целевого аспекта back-end'а в автоматическом режиме систематически генерировать тестовые наборы с целью достижения выбранного критерия полноты.

5.5. Тестирование

В рамках метода OTK оракул для тестирования back-end'а в компиляторе автоматически проверяет сохранение семантики выполнения программы во время обработки back-end'ом. Для этого в качестве тестовых воздействий меппером строятся такие программы, семантика выполнения которых полностью представляется их трассой. Такое свойство тестов позволяет свести

задачу проверки сохранения семантики к сравнению трассы с некоторой эталонной трассой.

Работа такого автоматического оракула заключается в следующем:

- каждый тест компилируется дважды – тестируемым компилятором и некоторым эталонным компилятором;
- обе откомпилированные версии запускаются на исполнение;
- полученные трассы сравниваются;
- считается, что тест прошел успешно в том и только том случае, если трассы эквивалентны.

6. Верификация подсистемы поддержки исполнения (runtime support)

Рассмотрим несколько примеров подсистем поддержки исполнения. Для низкоуровневых языков, таких как Си и Си++, поддержка исполнения ограничивается тем, что **компилятор включает в состав своей поставки стандартные библиотеки этих языков**.

Компилятор Java переводит исходный текст на языке Java в байт-код. Система поддержки времени исполнения включает в себя виртуальную машину Java, стандартную библиотеку классов Java SDK, в том числе, внутренние классы, обеспечивающие доступ к ресурсам операционной системы (ввод-вывод, потоки, и т.д.), набор средств взаимодействия Java-среды с окружением через JNI.

Другой пример относится к языку спецификации тестов TTCN3. Все известные в настоящее компиляторы TTCN3 построены как трансляторы – исходный текст на языке TTCN3 преобразуется в набор файлов на целевом языке (Си или Java), которые затем компилируются соответствующими компиляторами. Система поддержки времени выполнения представляет собой набор библиотек на целевом языке (Си или Java), которые реализуют встроенные возможности языка (таймеры, порты обмена сообщениями, стандартные функции и т.п.), предоставляют интерфейсы расширения (TCI и TRI) как сгенерированному коду, так и окружению, и, в зависимости от поставщика, различные дополнительные возможности.

Задачу верификации подсистемы поддержки исполнения можно разделить на следующие **подзадачи**:

1. Верификация программного интерфейса, предоставляемого сгенерированному коду.
2. Верификация встроенных возможностей языка и динамической семантики операций, предоставляемых подсистемой поддержки исполнения.

3. Верификация интерфейсов взаимодействия с окружением. Эта задача может быть разделена на две подзадачи:
   a. Верификация интерфейсов, предоставляемых окружению для взаимодействия с исполняемой системой.
   b. Верификация интерфейса, который предоставляет подсистема поддержки исполнения в исходном языке.

Рассмотрим эти задачи на примере транслятора TTCN3, который преобразует исходный текст TTCN3 в набор файлов на языке Си. В этом случае задачи верификации можно сформулировать следующим образом:

1. Верифицировать библиотеки непосредственно через предоставляемый ими интерфейс Си/Си++.
2. Верификация встроенных возможностей языка заключается в том, чтобы проверить выполнение требований динамической семантики TTCN3 в той части, которая реализована в подсистеме поддержки исполнения – проверка требований к передаче сообщений через порты обмена сообщениями, обработке событий в инструкциях выбора alt, запуску и остановке компонентов и т.д.
3. Верификация интерфейсов расширения заключается в проверке выполнения требований к стандартным интерфейсам расширения:
   a. Обращения к «скомпилированному тестовому набору» (Compiled Test Suite) извне через стандартные интерфейсы TCI и TRI, представленные в виде набора функциональных вызовов в программе на Си.
   b. Обращения к окружению скомпилированного тестового набора, представленные инструкциями и выражениями в исходных текстах на TTCN3, через стандартные интерфейсы TTCN3.

Аналогичные задачи можно сформулировать и для подсистем поддержки времени исполнения языка Java.

1. Верификация библиотеки поддержки исполнения. Необходимо учитывать, что почти все сервисы, предоставляемые поддержкой времени исполнения, представлены как классы Java, поэтому в данном случае верификация представляет собой тестирование Java-классов.
2. Верификация встроенных возможностей языка (например, синхронизации доступа) и динамической семантики (например, создания и уничтожения объектов).
3. Верификация интерфейсов расширения:

a. Обращения из Java-кода к ресурсам операционной системы через специализированные классы (потоки ввода-вывода, потоки исполнения Thread, класс System и т.д.), обращения из Java к внешним библиотекам, разработанным на других языках (JNI).

b. Верификация программного интерфейса виртуальной машины Java и доступа к объектам Java через JNI из программ на Си.

6.1. Методы верификации

Верификация встроенных возможностей языка основывается на методе ОТК (см. предыдущий подраздел). Целевые конструкции моделируются на формальном языке TreeDL, из формальной модели генерируются тестовые программы. Оракул строится на основе сравнения трасс исполнения тестовой программы, откомпилированной тестируемым компилятором, с трассой выполнения программы, откомпилированной на эталонном компиляторе.

Далее мы будем рассматривать только вопросы тестирования библиотек, предоставляющих поддержку времени исполнения.

Верификация библиотек времени исполнения основывается на методе функционального тестирования с использованием технологии автоматизированного тестирования UniTESK.[4].

Рассмотрим основные особенности UniTesK в приложении к тестированию программных интерфейсов:

- разделение построения тестовых воздействий и проверки правильности поведения целевой системы: тестовые воздействия строятся в *тестовых сценариях,* а проверка правильности поведения целевой системы осуществляется в *тестовых оракулах*;
- автоматизированное построение тестовых воздействий;
- представление функциональных требований к целевой системе в виде формальных спецификаций;
- автоматическая генерация тестовых оракулов из спецификаций;
- язык описания формальных спецификаций и тестовых воздействий «близок» к языку, на котором разработана целевая система;
- автоматически генерируются критерии качества покрытия требований;
- автоматически производится оценка качества покрытия требований при прогоне тестов.

Метод тестирования с использованием UniTESK подробно рассмотрен в препринте [9].

Для работы с формальными спецификациями и спецификациями тестов разработаны пакеты инструментов CTesK [10] и JavaTESK [12].

Представленный метод использовался при тестировании системы поддержки исполнения компилятора Sun Java 1.4.

6.2. Регламентирующие документы. Анализ требований

В рамках предлагаемого подхода под верификацией подсистемы поддержки времени исполнения понимается проверка соответствия этой подсистемы функциональным требованиям.

Функциональные требования, как правило, извлекаются из спецификации или стандарта языка или описания эталонной реализации языка. Так, в случае языка TTCN3 функциональные требования представлены в частях 1, 4, 5 и 6 спецификации языка [10]. Требования к подсистеме поддержки исполнения Java приводятся в Java Language Specification, спецификаций Java SDK и Java Native Interface, опубликованных фирмой Sun.

6.3. Моделирование

Требования к программному интерфейсу формализуются как контрактные спецификации. Такой подход к спецификации основан на представлении об описываемой системе как наборе компонентов, взаимодействующих с окружением и между собой. Контрактная спецификация состоит из *модельного состояния*, которое моделирует внутреннее состояние системы поддержки исполнения, и набора *спецификационных функций*. Спецификационные функции формализуют требования к обработке входящих и исходящих функциональных вызовов посредством логических выражений в постусловиях. Такой подход к спецификации программных интерфейсов близок к логике Т. Хоара и концепции «проектирования на основе контрактов» Б. Майера.

Формальные спецификации программного интерфейса записываются на расширении языков программирования Си [10] или Java [12]. Из формальных спецификаций автоматически генерируются *тестовые оракулы* – процедуры, которые выносят вердикт о соответствии наблюдаемого поведения системы спецификации.

6.4. Извлечение тестов

Тесты представляют собой конечные автоматы. С каждым переходом в автомате сопоставляется выдача тестового воздействия. Для текстового задания таких автоматов в используемых расширениях языков программирования предусмотрены специальные конструкции (*тестовые сценарии*, *сценарные функции* и *итераторы*), которые позволяют описывать автоматы в компактном виде.

6.5. Тестирование

При обходе автомата теста автоматически выдаются тестовые воздействия, связанные с текущим переходом автомата, собираются ответы целевой системы и выносится вердикт о соответствии наблюдаемого поведения системы её спецификации. Критерий полноты тестирования формулируется в терминах покрытия спецификации – должны быть покрыты все ветви функциональности, представленные в спецификационных функциях.

В ходе тестирования генерируется трасса теста, по которой генерируются отчёты. Отчёты содержат информацию о том, какие функции целевой системы тестировались и были ли выявлены в ходе тестирования расхождения со спецификацией.

7. Заключение

В работе представлен систематический подход к верификации компиляторов. Общая задача верификации компилятора декомпозирована на подзадачи верификации отдельных аспектов функциональности компилятора. Декомпозиция задачи верификации позволяет начинать верификацию на ранних этапах создания компилятора и проводить её параллельно разработке. Тем самым, сокращается продолжительность цикла разработки компилятора и снижаются суммарные затраты на тестирование компилятора.

Для всех выделенных подзадач предложены методы верификации, построенные в рамках единого подхода тестирования на основе формальных спецификации и моделей. Для всех методов существуют их подробные описания и инструментальная поддержка для их практического применения.

Представленный подход использовался при тестировании компиляторов Си и Java, а также трансляторов расширений языков программирования, разработанных в ИСП РАН [4].

Литература

[1] CMMI for Systems Engineering/Software Engineering, Version 1.02 (CMMI-SE/SW, V1.02) CMU/SEI-2000-TR-018 ESC-TR-2000-018 November 2000, P. 598

[2] A .Aho, R. Sethi, J. D. Ullman. *Compilers. Principles, Techniques, and Tools* // Addison-Wesley Publishing Company, Inc. – 1985.

[3] М.В. Архипова. Генерация тестов для семантических анализаторов // Вычислительные методы и программирование. 2006, том.7, раздел 2, 55–70.

[4] А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. Подход UniTesK к разработке тестов: достижения и перспективы. // Труды ИСП РАН, Москва, 2004, т. 5, 121–156.

[5] А.В. Демаков. TreeDL: язык описания графовых структур данных и операций над ними. // Вычислительные методы и программирование. 2006, том 7, раздел 2, 117–127.

[6] С.В. Зеленов, С.А. Зеленова. Генерация позитивных и негативных тестов для парсеров // Программирование, том. 31, №6, 2005, 25–40.

[7] С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. Применение модельного подхода для автоматического тестирования оптимизирующих компиляторов. [HTML] http://www.citforum.ru/SE/testing/compilers/

[8] С.В. Зеленов, Д.В. Силаков. Автоматическая генерация тестовых данных для оптимизаторов графических моделей. // Труды ИСП РАН, Москва, 2006, т. 9, 129–141.

[9] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. Препринт. М.: ИСП РАН, 2006. 50 с.

[10] CTesK - инструмент для тестирования программного обеспечения, реализованного на языке C. [HTML] http://www.unitesk.ru/content/category/5/13/32/

[11] ETSI ES 201 873-1 V3.1.1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. Sophia-Antipolis, France: ETSI, 2005. 210 c.

[12] JavaTESK - инструмент для тестирования программного обеспечения, реализованного на языке Java. [HTML] http://www.unitesk.ru/content/category/5/25/60/

[13] OTK – инструмент для тестирования программных систем, работающих с данными, имеющими сложную структуру. [HTML] http://www.unitesk.ru/content/category/5/15/34/

[14] SynTESK – инструмент для тестирования синтаксических анализаторов (парсеров) формальных языков. [HTML] http://www.unitesk.ru/content/category/5/16/35/.