УДК 004.052.2

ОПРЕДЕЛЕНИЕ МЕТРИКИ ДИВЕРСИФИЦИРОВАННОСТИ МУЛЬТИВЕРСИОННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА УРОВНЕ АЛГОРИТМОВ

Грузенкин Д.В., Якимов И.А., Кузнецов А.С., Царев Р.Ю.

ФГАОУ ВО «Сибирский федеральный университет», Красноярск, e-mail: gruzenkin.denis@good-look.su

Мультиверсионное программирование является одним из подходов к повышению уровня надежности программного обеспечения и его устойчивости к сбоям во время работы. Мультиверсионное программное обеспечение предполагает реализацию ряда версий его модулей. При этом данный подход тем эффективнее, чем выше диверсификация версий модулей мультиверсионного программного обеспечения. Статья посвящена исследованию меры диверсификации или различия между версиями модулей мультиверсионного программного обеспечения на уровне алгоритмов. Представлены результаты проверки гипотезы о наличии количественной метрики, определяющей степень различия между программами. Проведен эксперимент, в ходе которого производился анализ работы нескольких реализаций программных алгоритмов внутренней сортировки числовых массивов. В процессе выполнения эксперимента для каждого алгоритма была построена трасса его выполнения, которая отображает путь работы алгоритма в многомерном пространстве. Как показывают результаты проведенных исследований, анализ на основе предложенной концепции полученных трасс выполнения реализованных алгоритмов выявляет как различия между разными алгоритмами, так и наличие сходства между модификациями одного алгоритма.

Ключевые слова: метрика различия программного обеспечения, мультиверсионное программное обеспечение, автоматическое тестирование, модульное тестирование, надёжность программного обеспечения

N-VERSION SOFTWARE DIVERSITY METRIC DEFINITION ON THE ALGORITHM LEVEL

Gruzenkin D.V., Yakimov I.A., Kuznetsov A.S., Tsarev R.Yu.

Siberian Federal University, Krasnoyarsk, e-mail: gruzenkin.denis@good-look.su

N-version programming is one of the approaches to ensure a high level of software reliability and its tolerance to faults while executing. N-version software includes a set of versions of its modules. The higher the diversification of versions of N-version software modules the more effective the suggested approach. The article concerns the problem of version diversification at the level of algorithms. In the paper we present a hypothesis verification that there is a quantitative metric determining some differences between the programs. The experiment including the analysis of several realizations of program algorithms for inner sorting of numerical arrays has been taken. During this experiment, each algorithm was provided with an execution trace reflecting the way of algorithm operation in multidimensional space. The results of the conducted studies show that analysis based on the proposed concept for obtaining execution traces of implemented algorithms can reveal both difference of algorithms and the similarity of modifications of the same algorithm.

Keywords: measure of software diversity, N-version software, automatic test, unit-test, software reliability

Существуют такие области человеческой жизнедеятельности, в которых низкий уровень надёжности программного обеспечения (ПО) предопределяет высокую вероятность отказа, результатом которого может стать урон здоровью и жизни людей, а также значительные финансовые издержки. К таким областям относятся: атомная энергетика, химическая, металлургическая, военная и космическая отрасли промышленности и т.п.

Одним из наиболее хорошо себя зарекомендовавших подходов к повышению надёжности программного обеспечения является методология мультиверсионного программирования [2].

Для обеспечения высокого уровня надёжности версии модулей должны быть диверсифицированы. Диверсификация ПО осуществляется на четырёх уровнях:

1. Уровень алгоритмов.
2. Уровень языков программирования.
3. Уровень средств разработки.
4. Уровень средств тестирования [13].

Для обеспечения диверсификации различных версий модулей мультиверсионного ПО (МВПО) их разработка выполняется независимыми друг от друга группами разработчиков [3]. Однако в [3] также указывается, что такой подход отличается высокой стоимостью. При этом считается, что при такой независимой разработке версии модулей являются диверсифицированными по умолчанию. Однако на практике это оказывается не всегда так.

Для определения степени диверсифицированности версий модулей МВПО на уровне алгоритмов авторами предлагается введение метрики, определяющей степень различия между версиями.

Исследовательская гипотеза

Поскольку версии модулей МВПО выполняют какие-либо вычислительные операции над данными, авторами данной работы была выдвинута гипотеза, что определение меры различия версий на уровне алгоритмов может осуществляться путём сравнения изменений данных, обрабатываемых модулем.

Проверка данной гипотезы была осуществлена на примере алгоритмов внутренней сортировки числовых массивов.

Зачастую поведение вычислительных математических алгоритмов не инвариантно, т.е. один и тот же алгоритм может вести себя по-разному при подаче ему на вход различных данных. В связи с этим для обеспечения чистоты эксперимента необходимо осуществить систематический обход различных путей выполнения каждой сравниваемой программной реализации алгоритма.

Для удовлетворения данного требования был применён тот же подход, что используется при автоматизированной генерации unit-тестов. Несмотря на то, что мультиверсионное ПО как объект для применения подхода автоматизированной генерации тестов является новой областью, за счет его использования становится возможным прохождение всех ветвей кода каждого алгоритма.

Построение дерева выполнения программного алгоритма

С некоторыми допущениями на базовом уровне выполнение программы состоит из последовательных вызовов инструкций присваивания, а также условных и безусловных переходов [11]. В совокупности различные варианты выполнения программы образуют дерево выполнения. Корень дерева выполнения представляет собой точку входа, каждый из листов – точку выхода программы или подпрограммы. Маршрут вдоль ветвей дерева выполнения задается теми входными данными, которые подаются на вход программе. Таким образом, для исследования различных вариантов работы программы необходимо иметь различные входные данные, для чего может быть построен генератор входных данных.

Стоит отметить, что для многих программ дерево выполнения может иметь бесконечный размер, также некоторые пути могут не рассматриваться из-за технических ограничений генератора входных данных. Таким образом, для определения полноты тестового набора вводится *критерий покрытия* кода [7]. В данном случае тестовый набор считается адекватным, если он покрывает все ветви в *графе потока* [6] целевой программы.

Одной из методик, применяемых для систематической генерации входных тестовых данных, являются динамические символьные вычисления [8–12]. При использовании динамических символьных вычислений программы части входных данных рассматриваются как символьные неограниченные входы, другие же рассматриваются как конкретные. Во время выполнения программы накапливаются ограничения над символьными входами, которые в дальнейшем применяются для исследования альтернативных путей.

Авторами был реализован инструмент, позволяющий отслеживать изменения значений, хранимых в памяти программы. В рамках проведённого исследования отслеживались изменения значений, входящих в состав массивов, подаваемых на вход алгоритмам сортировки. Изменения отслеживались при входе в каждый новый базовый блок и при выходе из подпрограммы [14]. При дальнейшем изложении под *шагом работы алгоритма* будем понимать прохождение базового блока, сопровождаемое выводом промежуточных результатов сортировки. Под *трассой* будем понимать совокупность изменений входных данных, произведенных за время работы алгоритма, это отличается от классического понимания трассы [12].

Методы анализа результатов эксперимента

С целью выполнения анализа, а также введения меры различия между алгоритмами (метрики) для предметной области алгоритмов сортировки было определено следующее:

1. Массив на каждом шаге его сортировки – точка в многомерном пространстве.

2. Каждые две точки в многомерном пространстве образуют вектор.

3. Совокупность изменений значений входного массива за время работы алгоритма образует трассу.

4. Если в течение нескольких шагов алгоритма значение сортируемого массива остаётся неизменным, точка остаётся на месте.

Таким образом, определение метрики различия между алгоритмами может быть осуществлено по нескольким признакам:

1) отношение длины сортируемого массива (мерности пространства точки) к количеству шагов, т.е. условная скорость прохождения трассы:

$$\upsilon = \frac{|S|}{|steps|},$$

где $|S|$ – мерность пространства точки, $|steps|$ – количество точек в трассе;

2) количество и продолжительность общих отрезков у двух трасс, определим степень схожести алгоритмов (D) как

$$D = \frac{|SV|}{|V|},$$

где $|V|$ – количество рёбер трассы 1, $|SV|$ – количество рёбер трассы 1, совпадающих с рёбрами трассы 2;

3) длина (продолжительность) трасс:

$$l = \sum_{i=1}^{|Trace|-1} \sqrt{p_i * p_{i+1}},$$

где l – длина всей трассы, $|Trace|$ – количество точек в трассе, p_i – текущая точка трассы, p_i – следующая после текущей точка трассы;

4) отношение длины прямого пути от начальной до конечной точки и длины всей трассы:

$$I = \frac{l_{forward}}{l},$$

где $l_{forward}$ – длина прямого пути от первой до последней точки трассы, l – длина трассы.

Важно отметить, что такой подход позволяет определить наличие клонов среди алгоритмов, а также алгоритмы, эквивалентные друг другу.

Под клоном обычно подразумевается какой-либо исходный код, который применяется без изменений в нескольких различных программах [1]. Так как в данной работе обсуждается диверсификация ПО на уровне алгоритмов, то понятие «клон» имеет иное значение.

Авторами принято считать клонами две программные реализации алгоритмов, трассы выполнения которых полностью совпадают, то есть если выполняются следующие условия:

1. $|Trace_1| = |Trace_2|, i = \overline{1,2}$,

где |Trace| – количество точек в трассе

$$\forall\ point_i^1 \in Trace_1 \text{ и } point_i^2 \in Trace_2,$$

$$point_i^1 = point_i^2, i = \overline{1, |Trace_n|},$$

где $Trace_n$ – сравниваемая трасса, $point_i^n$ – текущая точка в n-й трассе, $|Trace_n|$ – количество точек в n-й трассе, $n = \overline{1,2}$, в случае, если условие 1 выполнено.

В данном контексте программные реализации алгоритмов считаются эквивалентными, если при подаче им на вход идентичных наборов данных они выдают идентичные результаты. Учитывать, являются ли две программные реализации алгоритмов клонами, имеет смысл только в случае их эквивалентности.

Особенности реализации

В данной работе использован генератор тестов [5], построенный по архитектуре EXE [8]. Для работы с генератором пользователю необходимо написать драйвер на языке Си. Генератор порождает тесты также на языке Си, которые далее компилируются и запускаются.

Для получения трасс разработана отдельная утилита для трансформации LLVM-IR кода [14]. Данным средством инструментируются операции выделения памяти в стеке, входные точки базовых блоков, а также точки выхода из функций. При распределении памяти в стеке утилита сохраняет указатель на хранимое значение и затем выводит его в нужном формате при входе в каждый базовый блок и выходе из функции, обеспечивая создание трасс в требуемом виде.

Результаты эксперимента

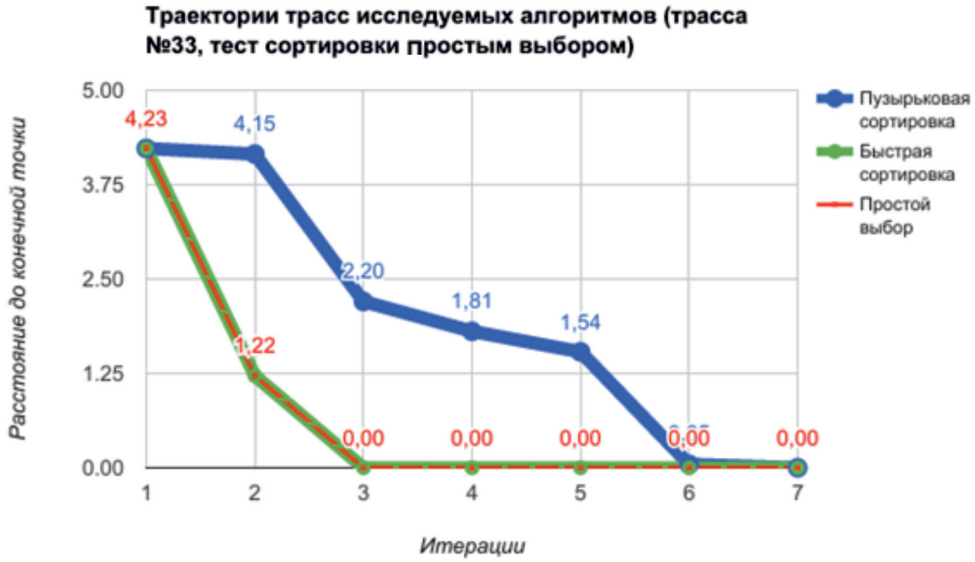
Для проведения эксперимента были написаны функции на языке С (Си), реализующие 3 алгоритма сортировки: пузырьковая сортировка, быстрая сортировка, простой выбор [4].

Для каждого алгоритма при помощи динамических символьных вычислений был порожден набор тестовых данных. Таким образом, было получено 3 тестовых набора для каждого из трех алгоритмов. В общей сложности было сгенерировано 104 тестовых случая: 24 – для пузырьковой сортировки, 47 – для быстрой сортировки и 33 – для простого выбора. При этом полученное покрытие кода соответствует как критерию *покрытия инструкций*, так и *покрытия переходов*. Размер символьного массива, подаваемого на вход, был ограничен четырьмя элементами.

Каждый алгоритм был запущен на трех тестовых наборах, таким образом, в результате было сгенерировано 9 наборов трасс для трех различных реализаций, т.е. в ходе эксперимента было получено и проанализировано 312 трасс. Это необходимо для оценки различий между ними [15]. Следует отметить, что каждому тестовому случаю в тестовом наборе соответствует определенная трасса. Полученные трассы были проанализированы с последующим вычислением метрик.

Результаты работы теста алгоритма простой выбор

Итерация	Метод пузырька		Быстрая сортировка		Простой выбор	
№	Координаты точки	Расстояние до конечной точки	Координаты точки	Расстояние до конечной точки	Координаты точки	Расстояние до конечной точки
1	{–1, –2, –7, –8}	4,23	{–1, –2, –7, –8}	4,23	{–1, –2, –7, –8}	4,23
2	{–2, –1, –7, –8}	4,15	{–8, –2, –7, –1}	1,22	{–8, –2, –7, –1}	1,22
3	{–2 ,–7, –1, –8}	2,20	{–8, –7, –2, –1}	0,00	{–8, –7, –2, –1}	0,00
4	{–2, –7, –8, –1}	1,81	{–8, –7, –2, –1}	0,00	{–8, –7, –2, –1}	0,00
5	{–7, –2, –8, –1}	1,54	{–8, –7, –2, –1}	0,00	{–8, –7, –2, –1}	0,00
6	{–7, –8, –2, –1}	0,05	{–8, –7, –2, –1}	0,00	{–8, –7, –2, –1}	0,00
7	{–8, –7, –2, –1}	0,00	{–8, –7, –2, –1}	0,00	{–8, –7, –2, –1}	0,00



Траектории трасс исследуемых алгоритмов

Поскольку представление результатов анализа всех 312 трасс является слишком объёмным и, как следствие, неприемлемым для приведения в данной статье, авторами было принято решение в качестве результатов эксперимента привести трассы для одного тестового случая.

Значения полученных трасс и расстояния каждой текущей точки до конечной представлены в таблице. На рисунке показаны траектории трасс для каждого из алгоритмов. Красным цветом (пунктиром) выделен график алгоритма, соответствующий запускаемому тесту.

На рисунке ось абсцисс соответствует количеству итераций. По оси ординат отмечаются расстояния от текущей до конечной точки работы алгоритма, которые определяются по формуле

$$l_i^{end} = \sqrt{point_{last} * point_{last}} - \\ -\sqrt{point_{last} * point_i}, i = \overline{1, |Trace|},$$

где $|Trace|$ – длина трассы, $point_{last}$ – конечная точка трассы, $point_i$ – текущая точка трассы.

Проанализируем результаты, полученные на основании запуска теста, сгенерированного для сортировки методом простого выбора.

1. Условная скорость прохождения трасс:

$$\upsilon_{bubble} = \frac{4}{7},$$

$$\upsilon_{quick} = \frac{4}{3} = 1\frac{1}{3},$$

$$\upsilon_{selection} = \frac{4}{3} = 1\frac{1}{3}.$$

2. Общие отрезки с другими трассами.

Трасса алгоритма пузырьковой сортировки не имеет общих отрезков с другими трассами, т.е. совпадает с ними на 0%.

Трассы алгоритмов быстрой сортировки и простого выбора совпадают друг с другом на 100%.

3. Длины трасс:

$$l_{bubble} = 57,694352552969804,$$

$$l_{quick} = 17,950274623911028,$$

$$l_{selection} = 17,950274623911028.$$

4. Показатели близости длины трасс к длине прямого пути:

$$I_{bubble} = 8,697750906394726,$$

$$I_{quick} = 2,706105718697761,$$

$$I_{selection} = 2,706105718697761,$$

при $l_{foeward} = 6,6332495807108$.

При анализе трасс всех тестовых случаев, включая приведенный выше, было выявлено, что:

1. Все алгоритмы эквивалентны, т.е. при подаче на вход одинаковых данных они выдают одинаковый результат.

2. Алгоритм пузырьковой сортировки значительно отличается от остальных алгоритмов, как по протяженности трассы и всех величин, вытекающих из неё, так и по количеству общих точек и отрезков.

3. Алгоритмы быстрой сортировки и простого выбора являются клонами, значения всех их показателей диверсифицированности совпадают.

Заключение

Таким образом, введение метрики различия ПО на уровне алгоритмов позволяет автоматически определять количественную степень различия на уровне алгоритмов между различными версиями модулей МВПО. При этом наличие спецификации не требуется – необходим только исходный код.

Кроме того, использование такой метрики может быть полезно, как на этапе разработки мультиверсионного программного обеспечения для оперативного внесения корректировок в разрабатываемый код, так и на этапе использования МВПО в готовом программном комплексе, например в процессе голосования, определяющего корректность результатов работы модулей.

Список литературы

1. Асланян А.К., Курмангалеев Ш.Ф., Варданян В.Г., Арутюнян М.С., Саргсян С.С. Платформенно-независимый и масштабируемый инструмент поиска клонов кода в бинарных файлах // Труды ИСП РАН. – 2016. – Т. 28, Вып. 5. – С. 215–226.

2. Завьялова О.И., Капулин Д.В., Царев Р.Ю. Минимизация межмодульного интерфейса при формировании мультиверсионного программного обеспечения // Системы управления и информационные технологии. – 2011. – № 3.1 (45). – С. 140–143.

3. Морозов В.А. Изоляция модулей мультиверсионного программного обеспечения на этапах разработки // Современные наукоемкие технологии. – 2007. – № 3. – С. 33–34.

4. Царев Р.Ю., Прокопенко А.В. Алгоритмы и структуры данных (CDIO): учебное пособие / Р.Ю. Царев // Сиб. федер. ун-т. – Красноярск, 2016. – 203 с.

5. Якимов И.А., Кузнецов А.С. Оптимизация читаемости тестов порождаемых при символьных вычислениях // Труды Института системного программирования РАН. – 2016. – Т. 28, Вып. 5. – С. 227–238.

6. Aho A.V., Sethi R., Ullman J.D. Compilers, principles, techniques, and tools // Boston: Addison-Wesley, 1986. – 1009 p.

7. Anand S., Burke E.K., Chen T.Y. An orchestrated survey of methodologies for automated software test case generation // Journal of Systems and Software. – 2013. – vol. 86. – № 8. – P. 1978–2001.

8. Cadar C., Ganesh V., Pawlowski P.M., Dill D.L., Engler D.R. EXE: automatically generating inputs of death // CCS '06 Proceedings of the 13th ACM conference on Computer and communications security (Alexandria, Virginia, USA, October 30 – November 03 2006). – P. 322–335.

9. Cadar C., Godefroid P., Khurshid S., Păsăreanu C.S. Symbolic execution for software testing in practice: preliminary assessment // ICSE '11 Proceedings of the 33rd International Conference on Software Engineering (Waikiki, Honolulu, HI, USA, May 21–28, 2011). – P. 1066–1071.

10. Cadar C., Sen K. Symbolic execution for software testing: three decades later // Communications of the ACM. – 2013. – vol. 56. – № 2. – P. 82–90.

11. Godefroid P., Klarlund N., Sen K. ACM SIGPLAN Notices // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. – 2005. – vol. 40. – № 6. – P. 213–223.

12. King J.C., Watson T.J. Symbolic Symbolic execution and program testing // Communications of the ACM. – 1976. – vol. 19. – № 7. – P. 385–394.

13. Kovalev I.V. System of Multi-Version Development of Spacecrafts Control Software. – Berlin: Pro Universitate Verlag Sinzheim, 2001. – 80 p.

14. LLVM Compiler infrastructure documentation [Электронный ресурс]. – URL: http://llvm.org/docs/index.html (дата обращения: 01.04.2017).

15. Ramos D.A., Engler D.R. Practical, low-effort equivalence verification of real code // CAV'11 Proceedings of the 23rd international conference on Computer aided verification (Snowbird, UT, July 14–20, 2011). – P. 669–685.