DOI: 10.15514/ISPRAS-2025-37(1)-6

Фреймворк автоматизации тестирования на гонки по данным

Е.А. Герлиц, ORCID: 0000-0002-1747-075X <gerlits@ispras.ru>
В.С. Мутилин, ORCID: 0000-0003-3097-8512 <mutilin@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В данной работе рассматривается класс параллельных программ над общей памятью и присущий этому классу программ тип ошибок – гонки по данным. Нами спроектирован тестовый фреймворк для разработки сценариев тестирования на гонки по данным по аналогии с широко применяемыми тестовыми фреймворками для последовательных программ. Основной проблемой при создании теста является недетерминизм, присущий выполнению многопоточных программ. С целью обеспечить повторяемость сценария тестирования в данной работе мы вводим понятие точек синхронизации в исходном коде программы, в которых тест регулирует порядок исполнения инструкций программы потоками при помощи внедрения синхронизационных действий. Разработанные тесты на гонки по данным выполняются полностью автоматически при помощи фреймворка и могут использоваться для регрессионного тестирования.

Ключевые слова: гонка по данным; состояние гонки; воспроизведение гонки по данным; тестирование на гонки по данным; многопоточная программа; параллельная программа с общей памятью; тестовый фреймворк; фреймворк автоматизации тестирования; синхронизация потоков.

Для цитирования: Герлиц Е.А., Мутилин В.С. Фреймворк автоматизации тестирования на гонки по данным. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 107–120. DOI: 10.15514/ISPRAS–2025–37(1)–6.

Towards a Test Automation Framework for Data Race Testing

E. Gerlits ORCID: 0000-0002-1747-075X <*gerlits@ispras.ru*>
V. Mutilin ORCID: 0000-0003-3097-8512 <*mutilin@ispras.ru*>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. In this paper, we examine shared memory concurrent programs and errors occurring in them, specifically data races. We design a test automation framework to develop data race revealing testing scenarios by analogy to test automation frameworks for sequential programs. The main problem complicating development of testing scenarios is the nondeterministic nature of multithreaded program executions. To provide repeatable testing scenarios we define a notion of synchronization points in the source code of the computer program where a test regulates execution of parallel threads with synchronization actions. Tests being developed with our test automation framework can be executed automatically and can be used for regression testing.

Keywords: data race; race condition; data race reproducing; data race testing; multithreaded program; shared memory concurrent program; testing framework; test automation framework; thread synchronization.

For citation: Gerlits E.A., Mutilin V.S. Towards a test automation framework for data race testing. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025. pp. 107-120 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-6.

1. Введение

В выполнении многопоточной программы присутствует гонка по данным, когда два потока обращаются к общей памяти без синхронизации, причём одно из обращений – это запись. Гонка по данным может приводить к некорректным вычислениям в программе и исключительным ситуациям. Поэтому задача проверки многопоточных программ на гонки по данным является актуальной.

Одним из возможных способов проверки является тестирование при помощи сценариев тестирования, программируемых вручную. Если проводить аналогию с тестированием последовательных программ, то этот способ тестирования реализуется при помощи тестовых фреймворков [1-3]. Однако если сценарии тестирования для последовательных программ создаются на основе функциональных требований к программе или её компонентам, то проверки многопоточных программ на гонки по данным на этапе функционального тестирования сравнительно редки на практике. Тем не менее они востребованы в иных случаях:

- В процессе циклической отладки [4-6] гонки по данным фактически создаётся тест, который воспроизводит выполнение программы с гонкой по данным – тест на реальную ошибку.
- При помощи этого теста проверяют отсутствие гонки по данным после исправления ошибки в исходном коде.
- Этот же тест может далее запускаться в ходе регрессионного тестирования, чтобы препятствовать проникновению гонки по данным в будущие версии программы.
- Код теста документирует основной результат отладки гонки по данным, которая может быть весьма трудоёмкой и длительной.

Таким образом, задача ручной разработки сценариев тестирования для выявления гонок по данным в многопоточных программах является актуальной.

Для автоматизации выполнения множества таких тестовых сценариев и получения результатов тестирования можно применять существующие тестовые фреймворки для последовательных программ. Однако их оказывается недостаточно:

- Многопоточным программам свойственен недетерминизм выполнения из-за непостоянства скорости исполнения инструкций ядрами процессора в многозадачной среде и других причин. В результате в повторном выполнении программы порядок исполнения потоками инструкций чтения и записи в память меняется. Как следствие, переменные получают иные значения, ход выполнения программы меняется и гонка по данным не возникает в повторном выполнении программы. Недетерминизм выполнения многопоточных программ мешает обеспечить основное свойство сценариев тестирования – повторяемость.
- Для осуществления проверок в тестах и вывода результатов этих проверок фреймворк предоставляет специальные функции – assert functions. Однако гонки по данным могут проявляться в виде ошибок или исключительных ситуаций не во всех выполнениях программы, в которых они происходят. Например, операция добавления элемента в список и операция проверки списка на пустоту, выполненные без синхронизации из разных потоков, не приводят к ошибке, если список изначально был не пуст. Разработка сценария тестирования, на котором гонка по данным проявляется в виде ошибки (обнаруживается при помощи assert functions), может быть намного более трудоёмкой задачей, чем разработка сценария тестирования, на котором гонка по данным обнаруживается более эффективными динамическими методами [7-10].

В данной работе мы предложим средства для решения обозначенных проблем и объединим эти средства с функциональностью существующих тестовых фреймворков, тем самым обеспечив возможность создания автоматически выполняемых тестов на гонки по данным по аналогии с тестами для последовательных программ. Разработанный нами фреймворк можно применять как в процессе циклической отладки гонки по данным для пошаговой разработки теста, воспроизводящего гонку по данным, так и для регрессионного тестирования на гонки по данным в системах непрерывной интеграции и развёртывания.

Работа устроена следующим образом. В разделе 2 мы приводим компонентную структуру фреймворка и последовательно её поясняем. Раздел 3 посвящён автоматизации выполнения тестов. В разделе 4 мы обсуждаем проблемы, которые возникли на практике в ходе разработки тестов при помощи фреймворка. Раздел 5 посвящён обзору связанных работ. Выводы по результатам исследования сделаны в разделе 6.

2. Фреймворк автоматизации тестирования на гонки по данным

На рис. 1 компонентная структура фреймворка для тестирования последовательных программ расширяется дополнительными компонентами (они выделены серым цветом), которые необходимы для построения тестов, нацеленных на гонки по данным.

2.1 Среда выполнения тестового фреймворка для последовательных программ

При наличии нескольких тестов на гонки по данным их последовательный запуск предлагается организовывать при помощи одного из существующих фреймворков для тестирования последовательных программ [1-3]. На рис. 1 показан случай, когда тесты компилируются вместе с кодом программы в один исполняемый образ. В этом случае за автоматизацию запуска тестов отвечает часть фреймворка для последовательных программ, которую мы называем средой выполнения фреймворка и которая также компилируется вместе с программой в один исполняемый образ. Выполнение каждого теста в свою очередь является последовательностью из трёх шагов:

1. *setup* – подготовка состояния программы для теста.

2. *test case* – многопоточный тестовый сценарий, в котором функции тестируемой программы вызываются из нескольких потоков, взаимодействующих через общую память.
3. *tear down* – перевод программы в начальное состояние для запуска следующего теста.

Рис. 1. Компонентная структура фреймворка для автоматизации выполнения тестов на гонки по данным.

Fig 1. Component structure of a framework automating execution of tests for data races.

2.2 Happens-before детектор

Тестовые фреймворки для последовательных программ предоставляют тестам функции для осуществления проверок и вывода (сохранения) результатов этих проверок – assert functions. При помощи этих функций гонки по данным можно обнаруживать по их проявлениям – ошибочным вычислениям или исключительным ситуациям.

В данной работе предлагается применять альтернативный способ обнаружения гонок по данным, а именно использовать один из существующих инструментов [4, 11] динамического мониторинга отношения happens-before [12] на множестве исполнений инструкций программы. Такой инструмент обнаруживает не последствия гонок по данным, а исполнения пар инструкций доступа к памяти без синхронизации, в которых один из доступов к памяти является записью и выполняется неатомарно.

Применение happens-before детектора вместо обнаружения гонки по данным по её проявлению облегчает разработку ручных сценариев тестирования так как:

- Количество выполнений программы, в которых два доступа к памяти, образующих гонку по данным, выполняются без синхронизации, не меньше количества выполнений, в которых гонка по данным проявляется в виде ошибки.
- Гонка по данным возникает до её проявления в виде наблюдаемой ошибки. Чтобы гонка по данным проявилась, требуется дополнительно организовать выполнение программы по определённому сценарию от момента её возникновения до момента её проявления.

Существуют альтернативные инструменты динамического обнаружения гонок по данным, которые могут быть более эффективны для конкретного класса программ. Так гонки по данным в операционной системе Linux ищут при помощи инструмента KCSAN [10].

2.3 Расстановка вызовов wait/notify функций

У недетерминированного выполнения многопоточной программы может быть несколько источников: непостоянная скорость исполнения инструкций программы ядрами процессора в многозадачной среде, вытесняющий планировщик потоков в операционной системе, использование случайных чисел, недетерминированное поведение внешних систем и системных вызовов и др. В данной работе ограничимся учётом только одного основного и непременного источника недетерминизма, которым является непостоянная скорость исполнения инструкций ядрами процессора.

Следствием непостоянной скорости исполнения инструкций ядрами процессора является изменение порядка исполнения инструкций программы потоками. Функции *wait* и *notify* используются для регулирования этого порядка. Основная идея состоит в следующем. Если требуется, чтобы некоторое исполнение инструкции i_1 в одном потоке произошло после некоторого исполнения инструкции i_2 в другом потоке, то в исходный код программы перед инструкцией i_1 вставляется вызов функции *wait,* которая приостанавливает выполнение текущего потока до уведомления, а после инструкции i_2 вызов функции *notify*, которая это уведомление отправляет.

Одна и та же инструкция программы может исполняться несколько раз разными потоками. Реализации функций *wait* и *notify* должны уметь определять целевые исполнения инструкций i_1 и i_2 соответственно, на которых они должны срабатывать. Принимать решение функции могут на основе аргументов, переданных им в качестве параметров, и внутреннего состояния (собственных глобальных переменных). Аргументами, в частности, могут быть значения глобальных и локальных переменных самой программы, доступные в местах вставки вызовов функций.

Каждый тест реализует некоторый набор *wait/notify* функций, вызовы которых вставляются в исходный код программы вручную. Так как согласно рис. 1 тесты компилируются вместе с программой в один исполняемый образ, то вызовы *wait/notify* функций необходимо добавить в исходный код программы сразу для всех тестов. Однако во время выполнения конкретного теста должны исполняться *wait* и *notify* функции только этого теста. Для этого положим, что тесты пронумерованы. Тогда при вставке вызова функции *wait* или *notify* в исходный код можно обрамлять вызов функции условным оператором в виде *if(test==n){wait/notify(…)}*, где *n* – номер теста, к которому относится этот вызов. Определение глобальной целочисленной переменной *test* должно быть частью среды выполнения фреймворка. Значение этой переменной устанавливает сам тест в начале своего выполнения в функции *setup*.

Основной проблемой является выявление мест в исходном коде, в которые следует добавить вызовы *wait/notify* функций, чтобы гонка по данным воспроизвелась. К сожалению, ручной способ расстановки *wait/notify* функций, как и всякий нетривиальный способ отладки, трудно алгоритмизировать и формально обосновать. На практике мы следовали следующему подходу:

1. Идентифицируем пару исполнений инструкций обращения к памяти, которые участвуют в гонке по данным. Будем называть эти исполнения инструкций целевыми. Если создаётся тест на существующую гонку по данным, ранее обнаруженную инструментом динамического анализа, то такой инструмент должен выдавать некоторую информацию о паре целевых исполнений инструкций (адреса инструкций, стеки вызовов функций во время их исполнения и др.).
2. Подобрать расстановку вызовов *wait/notify* функций, которая стабильно воспроизводит целевые исполнения инструкций.
3. Если happens-before детектор не сообщает о гонке по данным между целевыми исполнениями инструкций, то устранить отношение порядка между ними,

скорректировав расстановку *wait/notify* функций.

4. Убедиться, что гонка по данным воспроизводится стабильно. Для этого можно воспользоваться предложенным нами фреймворком, чтобы организовать многократные запуски теста с автоматической проверкой обнаружения целевой гонки по данным в каждом запуске.

3 Автоматизация добавления вызовов wait/notify функций в исходный код программы

Функции *wait/notify* выполняются только во время тестирования, поэтому обращения к ним, расставленные в исходном коде программы вручную, являются вспомогательным кодом. С увеличением количества тестов исходный код программы загромождается вспомогательным кодом. Это негативно сказывается на сопровождении исходного кода программы. Также некоторые стандарты безопасной разработки, такие как DO-178C [13], не допускают наличие исходного кода, который никогда не выполняется в реальных (не тестовых) запусках.

Автоматическое инструментирование исходного кода программы вызовами *wait/notify* функций позволяет устранить эту проблему. Предлагается следующий подход:

- В каждом тесте в некотором виде задаётся отношение между вызовами *wait/notify* функций (буквально текстовыми строками, содержащими вызов *wait/notify* функции со списком фактических параметров) и местами в исходном коде, в которые их нужно вставить.
- Компонент фреймворка проходит по всем тестам, получает от них вышеобозначенные отношения и вставляет вызовы *wait/notify* функций в указанные места в исходном коде.

***Определение 1**. Точка синхронизации (ТС) – это место в исходном коде программы для вставки вызовов wait/notify функций.*

Точка синхронизации может быть указана по-разному. В простейшем случае в качестве идентификатора точки синхронизации (ИТС) можно взять пару *(p, l),* где $p: \mathbb{N} \to \Theta$ – путь к модулю исходного кода в файловой системе, $l \in \mathbb{N}$ – номер строки, Θ – некоторое множество символов – алфавит, из которого составляются строки, например, пути в файловой системе, исходный код программы и др.

Исходный код программы может изменяться во время разработки и сопровождения, а идентификатор в виде пары *(p, l)* не устойчив ко многим этим изменениям. Чтобы повысить устойчивость, мы задаём точки синхронизации вручную в исходном коде в виде комментариев специального вида */*prefix n*/*, где *prefix* – некоторая строка символов, одинаковая для всех комментариев специального вида и позволяющая отделить их от других комментариев, а *n* – номер точки синхронизации, и нумеруем точки синхронизации локально в пределах функции, метода, класса.

***Определение 2**. Идентификатор точки синхронизации (ИТС) – это четвёрка (p, c, f, n), где p, c, f:* $\mathbb{N} \to \Theta$, $n \in \mathbb{N}$, Θ *– алфавит:*

- *p – путь в файловой системе к модулю исходного кода;*
- *c – имя класса;*
- *f – сигнатура функции или её часть, например, имя функции;*
- *n – порядковый номер точки синхронизации.*

Если *c* и *f* не заданы, то точка синхронизации установлена для модуля исходного кода. Иначе если *f* не задана, то точка синхронизации установлена для класса. Иначе если *c* не задан, то точка синхронизации установлена для функции вне класса. Иначе точка синхронизации установлена для метода.

Указанный метод идентификации точек синхронизации удобен для совместной работы, потому что локальная нумерация точек синхронизации позволяет без конфликтов параллельно нумеровать различные сущности (различные функции, методы, классы, структуры и файлы).

Отметим, задаваемый таким образом идентификатор точки синхронизации может оказаться не уникальным. Например, в случае вложенных (nested) одноимённых классов или функций, расположенных в одном модуле исходного кода. Но такие случаи редки и могут быть легко устранены – мы просто ведём общую нумерацию у одноимённых сущностей в пределах одного модуля исходного кода.

Определение 3. *Синхронизационное действие (СД) – это исходный код вызова функций wait или notify с указанием фактических параметров.*

Указание фактических параметров означает, что синхронизационное действие предполагает его вставку в конкретную точку синхронизации. Соответствующее отношение {ТС} → {СД} между множеством точек синхронизации в программе и множеством синхронизационных действий теста задаётся в тесте вручную путём перечисления пар (*ИТС, СД)* в отдельном текстовом файле.

Компонентная структура фреймворка изображена на рис. 2. Она расширяет компонентную структуру фреймворка из рис. 1 автоматическим инструментированием программы синхронизационными действиями тестов.

Рис. 2. Компонентная структура фреймворка для тестирования на гонки по данным с автоматическим инструментированием исходного кода вызовами wait/notify функций.

Fig 2. Component structure of a framework for data race testing which includes automatic source code instrumentation with wait/notify function calls.

3.1 Алгоритм инструментирования исходного кода программы синхронизационными действиями тестов

Псевдокод алгоритма представлен на листинге 1. В алгоритме используются следующие обозначения:

- *S* – множество всевозможных строк $\mathbb{N} \to \Theta$, где Θ – множество символов (алфавит).
- *ID* – множество всевозможных идентификаторов точек синхронизации.

Алгоритм предполагает, что некоторый скрипт предварительно прошёл по каталогу с тестами и в директории каждого теста прочитал файл с отношением {ТС} → {СД}, интерпретировал его и сформировал для алгоритма аргумент *sync* – отношение между номерами тестов и функциями, отображающими идентификаторы точек синхронизации на синхронизационные

действия теста. Аналогично сформирован аргумент *header* – отношение между номерами тестов и заголовочными файлами (путями к ним в файловой системе), в которых объявлены *wait/notify* функции, используемые тестом. Аргумент *src* моделирует содержимое инструментируемых файлов исходного кода программы в виде функции, которая путям к файлам в файловой системе ставит в соответствие функцию, отображающую номера строк этих файлов на исходный код в этих строках.

Вход: $header : \mathbb{N} \to S$ {номер теста -> путь к заголовочному файлу теста}
$sync : \mathbb{N} \to (ID \to S)$ {номер теста -> (ИТС -> СД)}
$src : S \to (\mathbb{N} \to S)$ {путь к файлу кода -> (номер строки -> код)}
Выход: src - инструментированный код

$a : \mathbb{N} \to S, a = \varnothing \to \mathbb{N}$ {номер СД -> код СД}
$t : \mathbb{N} \to \mathbb{N}, t = \varnothing \to \mathbb{N}$ {номер СД -> номер теста}
$i : ID \to \mathbb{N}, i = \varnothing \to \mathbb{N}$ {ИТС -> номер СД}
$f : S \to ID, f = \varnothing \to ID$ {путь к файлу кода программы -> ИТС}
for all $tid \in Dom(sync)$ **do**
  for all $sid \in Dom(sync(tid))$ **do**
    $n = |Dom(a)| + 1$
    $a = a \cup \{(n, sync(tid)(sid))\}$
    $t = t \cup \{(n, tid)\}$
    $i = i \cup \{(sid, n)\}$
    $f = f \cup \{(sid.f, sid)\}$
for all $fpath \in Dom(f)$ **do**
  $fsid = \{sid \in Im(f) : (fpath, sid) \in f\}$
  $l : ID \to \mathbb{N}, l = SidToLine(fsid, fpath)$ {ИТС -> номер строки}
  $m = 0$
  $h = \varnothing$
  while $|Dom(l)| > 0$ **do**
    $(sid, line) \in l : \forall l_2 \in Im(l) : line <= l_2$ {Выбор ТС по порядку}
    $l = l \backslash (sid, line)$
    for all $n \in Im(i) : (sid, n) \in i$ **do**
      $code = Concat(\text{“}if(test ==\text{”}, Itoa(t(n)), \text{“})\{\text{”}, a(n), \text{“}\}\backslash n\text{”})$
      $src(fpath) = Insert(src(fpath), line + m, code)$
      $h = h \cup \{hpath : (t(n), hpath) \in header\}$
      $m = m + 1$
  for all $hpath \in h$ **do**
    $code = Concat(\text{“}\#include <\text{”}, hpath, \text{“}> \backslash n\text{”})$
    $src(fpath) = Insert(src(fpath), 1, code)$

Листинг 1. Алгоритм инструментирования исходного кода программы синхронизационными действиями тестов.
Listing 1. Algorithm for program source code instrumentation with the test synchronization actions.

Алгоритм инструментирования вызывает ряд тривиальных вспомогательных функций, которые мы не будем приводить:

- *Concat*: $(\mathbb{N} \to \Theta) \times (\mathbb{N} \to \Theta) \longrightarrow (\mathbb{N} \to \Theta)$ – конкатенация последовательностей (строк).
- *Itoa*: $\mathbb{N} \longrightarrow (\mathbb{N} \to \{\text{´0´}, ..., \text{´9´}\})$ – перевод числа в строку.
- *Insert*: $(\mathbb{N} \to \Theta) \times \mathbb{N} \times \Theta \longrightarrow (\mathbb{N} \to \Theta)$ – вставка элемента из множества Θ в последовательность $\mathbb{N} \to \Theta$ перед элементом, позиция которого задаётся целым из множества $\mathbb{N}$.

Функция *SidToLine* выполняет поиск номеров строк, в которых располагаются точки синхронизации. На вход функция принимает множество *fsid* идентификаторов точек синхронизации, которые находятся в одном модуле исходного кода программы. Для идентификаторов вида $(p, l) : p \in S, l \in \mathbb{N}$, где p – путь к файлу, а l – номер строки, алгоритм функции *SidToLine(fsid, fpath)* представлен на листинге 2. Точный алгоритм поиска номеров строк для идентификаторов точек синхронизации по определению 2 предполагает синтаксический анализ модуля исходного кода, в котором находятся точки синхронизации. Краткое описание такого алгоритма приведено в практической части работы в разделе 4.3.

```
function SidToLine(fsid, fpath)
  if ∃sid ∈ fsid then
    {(sid, sid.l)} ∪ SidToLine(fsid\{sid}, fpath)
  else
    ∅
```

Листинг 2. Псевдокод функции SidToLine
Listing 2. Pseudocode of SidToLine function

4 Результаты практического применения фреймворка

Разработанный нами фреймворк, в частности, применялся в проекте по созданию открытой распределённой операционной системы для умных устройств. При помощи фреймворка было разработано несколько регрессионных тестов на гонки по данным, обнаруженные инструментом TSan. Реализация алгоритма инструментирования исходного кода синхронизационными действиями тестов позволила запускать указанные тесты автоматически на любой версии кода программы по изменению (commit) в системе непрерывной интеграции и развёртывания.

По результатам практического применения фреймворка выявлено две проблемы:

1. Наведение частичного порядка между исполнениями инструкций программы из-за синхронизации внутри *wait/notify* функций.
2. Взаимная блокировка потоков внутри *wait* функций.

Возникла также необходимость идентификации точек синхронизации по определению 2, так как указанный способ идентификации точек синхронизации устойчив к некоторым изменениям исходного кода: добавление новых сущностей, удаление существующих сущностей, изменение и перемещение сущностей, не содержащих точки синхронизации, и др. Эта устойчивость позволяет запускать тесты на множестве версий программы с относительно небольшими изменениями, как это происходит в случае автоматических запусков тестов в системах непрерывной интеграции и развёртывания.

4.1 Наведение частичного порядка между инструкциями программы

Ожидание *wait* одним потоком уведомления *notify* от другого предполагает взаимодействие через общую память. Чтобы это взаимодействие не приводило к гонкам по данным, в реализациях *wait/notify* функций необходимо применять средства синхронизации потоков или атомарные операции. Однако ввиду транзитивности отношения happens-before, синхронизация потоков, инициированная *wait/notify* функциями, наводит порядок между исполнениями инструкций самой программы. Чтобы этого не происходило можно помечать исходный код (*wait/notify* функции) специализированными аннотациями, которые предоставляет happens-before детектор, чтобы исключить этот код из наблюдения. Для языка C++ также можно использовать атомарные операции с указанием порядка std::memory_order_relaxed [14].

4.2 Взаимная блокировка потоков

Данная проблема особенно актуальна, если автоматический тест на гонку по данным используется для регрессионного тестирования. В течение жизненного цикла программы её код исправляется и дорабатывается. В результате может возникнуть версия программы, инструментирование которой синхронизационными действиями теста может привести к взаимной блокировке потоков, то есть к ситуации, когда два потока выполняют функции *wait*, бесконечно ожидая уведомление друг от друга. Чтобы эту ситуацию обнаружить, можно либо установить глобальный таймаут для теста, либо выполнять ограниченное по времени ожидание в функциях *wait* с уведомлением о таймауте.

4.3 Поиск номеров строк для точек синхронизации

Получить номера строк для идентификаторов точек синхронизации по определению 2 можно следующим способом. Перед инструментированием выполнить вспомогательную сборку программы с автоматическим перехватом опций компилирования файлов. Сохранить опции компилирования в хранилище (базе данных или файле). Для этого можно применить, например, инструмент Clade [15]. Реализовать функцию *SidToLine*, которая:

1 Извлекает опции компилирования файла *fpath* из хранилища.

2 Использует эти опции для трансляции файла *fpath* в абстрактное синтаксическое дерево (АСД), которое можно анализировать программно. Для исходного кода на языке C++ эта задача может быть решена, к примеру, при помощи библиотеки libclang [16].

3 Для каждого идентификатора *sid* из *fsid*:

3.1 По идентификатору *sid* находит в АСД объект комментарий *obj*.

3.2 От объекта *obj* получает номер строки *end*, в которой *obj* завершается.

3.3 Добавляет пару *(sid, end+1)* к отношению, возвращаемому функцией.

5. Обзор связанных работ

Если фреймворк, предложенный в настоящей работе, нацелен на разработку тестовых сценариев вручную, то фреймворк, предложенный в работе [17], нацелен на автоматический отбор подмножества тестов для регрессионного тестирования. Фреймворк выявляет общие переменные программы, то есть переменные, к которым выполняется доступ из нескольких потоков программы. Для этих целей применяется статический анализ escape analysis [18-19]. Далее фреймворк определяет подмножество общих переменных, которые затронуты изменениями в программе. На следующем шаге отбирается подмножество тестов, которые ищут гонки по данным между обращениями к этим переменным. На последнем шаге фреймворк упорядочивает тесты по убыванию количества общих переменных, обращения к которым проверяются на гонки по данным.

В качестве альтернативы ручной разработке сценария тестирования на конкретную гонку по данным можно рассматривать метод автоматического поиска (подбора) сценария тестирования, который эту гонку по данным воспроизводит (обнаруживает). Так в работе [20] инструмент CHESS отслеживает события в программе, такие как обращения к сервисам операционной системы, которые используются для организации многопоточных вычислений, и на основе наблюдений строит модель выполнения программы в виде happens-before графа [12]. Эта модель предполагает последовательную консистентность вычислений [21]. Из модели вычисления инструмент извлекает информацию о потенциально возможных альтернативных последовательностях переключений потоков – расписаниях. Выполнение программы по выбранному расписанию организуется при помощи управления планировщиком потоков. Параллельно с регулированием выполнения программы

инструмент CHESS отслеживает возникающие ошибки, в частности, гонки по данным. Если при выполнении программы по некоторому расписанию обнаружена ошибка, то инструмент помечает (сохраняет) это расписание, чтобы выполнение с ошибкой можно было повторить с целью отладки.

Ручная разработка сценариев тестирования на гонки по данным подразумевает итеративный подход, когда в процессе разработки тест запускается, чтобы на основе анализа (отладки) его выполнения понять, как тест должен быть доработан, например, поправлена расстановка *wait/notify* функций. Итеративному сценарию мешает недетерминизм выполнения многопоточных программ, из-за которого два последовательных запуска одной и той же версии теста приводят к существенно различным выполнениям программы. Чтобы эту проблему преодолеть, можно применять методы записи и воспроизведения выполнения программ. Так метод Instant Replay, изложенный в работе [4], моделирует все способы межпоточного взаимодействия как операции над разделяемыми объектами в общей памяти. На этапе мониторинга метод Instant Replay сохраняет информацию о событиях чтения и записи в разделяемые объекты, а на этапе воспроизведения на основе этой информации организует повторное выполнение программы таким образом, чтобы на каждом шаге все потоки читали те же данные, которые они читали на этапе мониторинга.

В нашем фреймворке для обнаружения гонок по данным используется инструмент динамического анализа – happens-before детектор. Метод happens-before не требует, чтобы гонка по данным реально происходила, то есть чтобы два доступа к памяти выполнялись одновременно либо один за другим. Метод happens-before обнаруживает, что два доступа к памяти выполняются без синхронизации (не упорядочены), один из них выполняет запись и один из них выполняется неатомарно. При этом, если инструмент не наблюдает какие-то события синхронизации, то он может выдавать ложные предупреждения о гонках по данным. Чтобы убедиться в реальности гонки по данным, можно продолжить уточнять сценарий тестирования до тех пор, пока гонка по данным фактически не произойдёт. В работе [22] решается схожая задача. На вход анализу подаётся трасса выполнения программы. В этой трассе при помощи метода lock-set [8] выявляется гонка по данным, то есть в трассе обнаруживаются два обращения к одному участку памяти из разных потоков, одно из них выполняет запись и пересечение множеств захваченных блокировок в момент обращений к памяти пусто. Далее по трассе и гонке по данным автоматически выполняется поиск возможного порядка исполнения инструкций программы потоками, на котором два обращения к памяти, участвующие в гонке по данным, происходят одно за другим, то есть воспроизводится гонка по данным. Этот поиск выполняется статически за счёт формулирования задачи поиска в виде формулы для SMT (Satisfiability modulo theories) [23] решателя [24].

В индустрии разработки программного обеспечения существуют инструменты, автоматизирующие функциональное тестирования программ. Некоторые подобные инструменты поддерживают разработку тестов для обнаружения состояний гонки в многопоточных программах. Примером является инструмент RapiTest [25]. Инструмент предлагает табличный формат для описания тестовых сценариев, а для регулирования порядка выполнения потоков предлагает вставлять задержки при помощи оператора wait (ожидание). Ошибки обнаруживаются при помощи традиционных проверок результатов вычислений.

6. Выводы

Разработка тестовых сценариев для обнаружения или воспроизведения гонок по данным зачастую представляет собой трудную задачу. Основной проблемой является недетерминизм, присущий выполнению многопоточных программ. По аналогии с тестированием

последовательных программ, чтобы облегчить разработку тестовых сценариев на гонки по данным, в работе предлагается применять тестовый фреймворк.

Описана компонентная структура такого фреймворка. Основные отличия от фреймворков для последовательных программ – это использование динамического happens-before анализа для обнаружения гонок по данным и регулирование порядка исполнения инструкций программы при помощи добавления в исходный код программы вызовов функций в парадигме wait/notify (ожидать/уведомить).

Разработанные при помощи фреймворка тесты на гонки по данным выполняются автоматически, поэтому могут применяться в системах непрерывной интеграции и развёртывания для регрессионного тестирования.

Список литературы / References

[1]. Beust C., Suleiman H. Next generation Java testing: TestNG and advanced concepts. – Pearson Education, 2007.

[2]. Massol V. JUnit in action. – 2004.

[3]. Hyuk Myeong. Googletest In Practice: Unit Testing Guide for C++ Programmers. – 2021.

[4]. Leblanc. Debugging parallel programs with instant replay //IEEE Transactions on Computers. – 1987. – Т. 100. – №. 4. – С. 471-482.

[5]. Wang Y. et al. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing //Proceedings of annual IEEE/ACM international symposium on code generation and optimization. – 2014. – С. 98-108.

[6]. Thane H., Hansson H. Using deterministic replay for debugging of distributed real-time systems //Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000. – IEEE, 2000. – С. 265-272.

[7]. Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice //Proceedings of the workshop on binary instrumentation and applications. – 2009. – С. 62-71.

[8]. Savage S. et al. Eraser: A dynamic data race detector for multithreaded programs //ACM Transactions on Computer Systems (TOCS). – 1997. – Т. 15. – №. 4. – С. 391-411.

[9]. Erickson J. et al. Effective {Data-Race} Detection for the Kernel //9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). – 2010.

[10]. The Kernel Concurrency Sanitizer. The Linux Kernel Organization. Available at: https://docs.kernel.org/dev-tools/kcsan.html, accessed 04.01.2025.

[11]. Adve S. V. et al. Detecting data races on weak memory systems //ACM SIGARCH Computer Architecture News. – 1991. – Т. 19. – №. 3. – С. 234-243.

[12]. Lamport L. Time, clocks, and the ordering of events in a distributed system //Concurrency: the Works of Leslie Lamport. – 2019. – С. 179-196.

[13]. RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification.

[14]. ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++.

[15]. Исходный код инструмента Clade для перехвата команд сборки. Веб-ссылка: https://github.com/17451k/clade, доступно 01.04.2024.

[16]. Документация компилятора Clang с главой, посвящённой библиотеке libclang. Веб-ссылка: https://clang.llvm.org/doxygen/index.html, доступно 01.04.2024.

[17]. Yu T., Srisa-an W., Rothermel G. SimRT: An automated framework to support regression testing for data races //Proceedings of the 36th international conference on software engineering. – 2014. – С. 48-59.

[18]. Halpert R. L., Pickett C. J. F., Verbrugge C. Component-based lock allocation //16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007). – IEEE, 2007. – С. 353 - 364.

[19]. Huang J., Liu P., Zhang C. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs //Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. – 2010. – С. 207-216.

[20]. Musuvathi M. et al. Finding and Reproducing Heisenbugs in Concurrent Programs //OSDI. – 2008. – Т. 8. – №. 2008.

[21]. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs //IEEE transactions on computers. – 1979. – Т. 100. – №. 9. – С. 690-691.

[22]. Said, M., Wang, C., Yang, Z., & Sakallah, K. (2011, April). Generating data race witnesses by an SMT-based analysis. In NASA Formal Methods Symposium (pp. 313-327). Springer, Berlin, Heidelberg.
[23]. De Moura L., Bjørner N. Satisfiability modulo theories: introduction and applications //Communications of the ACM. – 2011. – Т. 54. – №. 9. – С. 69-77.
[24]. Dutertre B., De Moura L. A fast linear-arithmetic solver for DPLL (T) //International Conference on Computer Aided Verification. – Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. – С. 81-94.
[25]. https://www.rapitasystems.com/products/rapitest.

Информация об авторах / Information about authors

Евгений Анатольевич ГЕРЛИЦ – научный сотрудник отдела технологий программирования ИСП РАН. Область научных интересов: методы контроля и обеспечения качества программного обеспечения, методы динамической верификации, статической верификации и анализа программ, формальные методы.

Evgeny Anatolievich GERLITS – researcher at the Software Engineering Department of ISP RAS. Main research interests: software quality control and assurance, dynamic and static software verification and analysis, formal methods.

Вадим Сергеевич МУТИЛИН – старший научный сотрудник Института системного программирования. Сфера научных интересов: статический и динамический анализ программ.

Vadim Sergeevich MUTILIN – senior researcher at the Software Engineering Department of the Institute for System Programming of the RAS. Main research interests: static and dynamic program analysis.