DOI: 10.15514/ISPRAS-2026-38(3)-7

Поиск модификаций коллекции во время её перечисления в исходном коде на языке C# методами статического анализа

[1,2] К.Х. Ханевская, ORCID: 0009-0000-8162-8243 <k.hanevskaja@ispras.ru>
[1,2] У.В. Тяжкороб, ORCID: 0000-0002-2375-6842 <tsiazhkorob@ispras.ru>
[1,3] В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>

[1] Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

[2] Московский физико-технический институт,
Россия, 141700, Московская область, г. Долгопрудный, Институтский пер., 9.

[3] Московский государственный университет имени М.В. Ломоносова,
Россия, 119991, Москва, Ленинские горы, д. 1.

Аннотация: Статья посвящена разработке метода выявления ошибок модификации коллекций во время их перечисления в языке C#. Подобные ошибки приводят к исключению InvalidOperationException и аварийному завершению программы во время выполнения. Предлагаемое решение реализовано в рамках промышленного статического анализатора SharpChecker и использует межпроцедурный символьный анализ, чувствительный к потоку управления и путям выполнения. Детектор отслеживает как прямые изменения коллекций в цикле foreach, так и модификации, происходящие через цепочки вызовов методов. Тестирование на проектах с открытым исходным кодом показало точность около 69%. Результаты демонстрируют эффективность подхода для обнаружения сложных случаев модификации коллекций.

Ключевые слова: статический анализ; символьное выполнение; поиск ошибок; модификация коллекции; перечисление коллекции.

Для цитирования: Ханевская К.А., Тяжкороб У.В., Игнатьев В.Н. Поиск модификаций коллекции во время её перечисления в исходном коде на языке C# методами статического анализа. Труды ИСП РАН, том 38, вып. 3, part 1, 2026 г., стр. 129–142. DOI: 10.15514/ISPRAS–2026–38(3)–7.

Detection of collection modifications during its enumeration in the C# source code using static analysis

[1,2] K.A. Khanevskaya, ORCID: 0009-0000-8162-8243 <k.hanevskaja@ispras.ru>
[1,2] U.V. Tsiazhkorob, ORCID: 0000-0002-2375-6842 <tsiazhkorob@ispras.ru>
[1,3] V.N. Ignatiev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>

[1] Ivannikov Institute for System Programming of the RAS,
25 Alexander Solzhenitsyn st., Moscow, 109004, Russia

[2] Moscow Institute of Physics and Technology
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia

[3] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. The article is dedicated to the development of the method for detecting collection modification error during their enumeration in C# programs. Such errors lead to an InvalidOperationException and program crash at runtime. The proposed solution is implemented within the industrial static analyzer SharpChecker and uses interprocedural symbolic analysis, sensitive to control flow and execution paths. The detector tracks both direct modifications of collections within a foreach loop and modifications occurring through chains of method calls. Testing on open-source projects showed an accuracy of about 69%. The results demonstrate the effectiveness of the approach for detecting complex cases of collection modification.

Keywords: static analysis; symbolic execution; bug detection; collection modification; collection enumeration.

For citation: Khanevskaya K.A., Tsiazhkorob U.V., Ignatiev V.N. Detection of collection modifications during its enumeration in the C# source code using static analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 1, 2026, pp. 129-142 (in Russian). DOI: 10.15514/ISPRAS-2026-38(3)-7.

1. Введение

Ошибки, связанные с изменением коллекции в процессе её перечисления, сложно выявить, поскольку они зачастую проявляются только при определённых сценариях выполнения программы. Изменение коллекции может приводить к возникновению исключения InvalidOperationException при получении перечислителя для ее следующего элемента. Существующие инструменты успешно выявляют лишь тривиальные случаи таких ошибок. Это актуально при интеграции в среду разработки, и чаще всего легко обнаруживается при тестировании. В то время как сложные сценарии, включающие цепочки вызовов методов и неочевидные условия выполнения, остаются за пределами возможностей таких инструментов и могут проявляться лишь в редких сценариях использования, не покрытых тестами. Отладка таких ошибок затруднена из-за сложности выявления и воспроизведения таких сценариев. Поэтому эффективным способом поиска таких ошибок является статический анализ исходного кода, не требующий запуска программы и входных данных для воспроизведения пути выполнения с ошибкой.

Согласно рейтингу популярности языков программирования TIOBE [1], C# стабильно занимает пятое место, что обуславливает высокую востребованность инструментов анализа для программ, написанных на нём. Это подтверждает актуальность разработки детектора таких ошибок.

Целью данной работы является разработка детектора ошибки модификации коллекции в процессе перечисления для языка C# в рамках промышленного статического анализатора SharpChecker. Для достижения поставленной цели был создан набор синтетических тестов, покрывающих различные случаи наличия ошибки в коде, разработан и реализован алгоритм работы детектора. В работе приводятся результаты его тестирования на реальных проектах с открытым исходным кодом и их сравнение с другими существующими инструментами,

оценивается эффективность предложенного подхода, доля истинных предупреждений и производительность детектора.

1.1 Мотивационный пример

Рассмотрим пример ошибки, основанной на реальной проблеме в коде. (листинг 1).

```
1  void AddElem(int elem, List<int> elems, bool flag)
2  {
3      if (flag)
4      {
5          elems.Add(elem); // добавление элемента в коллекцию
6      }
7  }
8  void ChangeSize(List<int> elems)
9  {
10     elems.Capacity += 1; // изменение размера коллекции
11 }
12 void Foo()
13 {
14     List<int>  elems = new List<int> () { 1, 2, 3 };
15     foreach (int elem in elems) // цикл по коллекции elems
16     {
17          AddElem(elem, elems, false); // вызов метода AddElem, в теле
которого содержится модификация коллекции elems
18         ChangeSize(elems); // вызов метода ChangeSize, в теле которого
содержится модификация коллекции elems
19     }
20 }
```

Листинг 1. Мотивационный пример, демонстрирующий ошибку модификации коллекции во время перечисления.

Listing 1. Motivational example demonstrating the error of modifying a collection during enumeration.

В этом примере показаны две операции, которые могут привести к модификации коллекции внутри цикла `foreach`. В теле цикла, осуществляющего обход по коллекции `elems`, происходят вызовы двух различных методов:

1. вызов метода `AddElem`, в теле которого происходит прямое изменение состава коллекции с помощью вызова метода `Add`, который добавляет новый элемент;
2. вызов метода `ChangeSize`, модифицирующий внутреннюю структуру коллекции через изменение свойства `Capacity`, определяющего вместимость коллекции.

Однако только второе действие приведёт к возникновению исключения. Первое является недостижимым из-за того, что переменная `flag` равна `false`, соответственно в данном случае модификация не будет произведена.

Этот пример построен путем упрощения реальной ошибки в коде для демонстрации необходимости поддержки детектором межпроцедурного анализа и чувствительности к путям выполнения.

1.2 Существующие решения

Проблема модификации коллекции во время перечисления является часто встречаемой ошибкой в программном коде и актуальна для различных языков программирования. Для языка Java в работе [2] представлен подход на основе механизма Tracematches, позволяющего специфицировать шаблоны некорректного использования объектов. Статический анализатор

выявляет нарушения, комбинируя потокочувствительный анализ состояний конечного автомата с межпроцедурной информацией о вызовах. Напрямую к теме данной работы относится приводимый в статье tracematch FailSafeIter, детектирующий модификацию коллекции в момент активной итерации. Основное ограничение данного подхода заключается в необходимости ручного описания шаблонов для каждого типа ошибки, что ограничивает способность анализатора обнаруживать ошибки, отличные от описанных в шаблонах.

Для C++ в работе [3] предложен комплексный подход, включающий проверки на этапе компиляции средствами метапрограммирования, которые эффективны для простых случаев, но не масштабируются на сложные межпроцедурные сценарии, а также статический анализ на основе синтаксического дерева компилятора Clang [4] и динамическую валидацию через скрипты отладчика GDB [5], требующую выполнения программы для обнаружения ошибок.

В промышленном статическом анализаторе Svace [6] реализованы детекторы такой ошибки для языков Java и Kotlin.

Для языка C# существуют детекторы рассматриваемой ошибки в различных промышленных анализаторах. К ним относится детектор V3221 [7] в анализаторе PVS-Studio [8], обнаруживающий изменение коллекции в цикле `foreach`, а также проверка «Possible 'System.InvalidOperationException: Collection was modified'» [9] в анализаторе ReSharper [10]. Анализ открытых возможностей детектора PVS-Studio показывает, что он может выявлять только те случаи модификации коллекций, которые происходят в пределах одного метода. Документация ReSharper не раскрывает детали реализации и возможности данной проверки, что не позволяет оценить её эффективность.

Таким образом, анализ существующих решений позволяет заключить, что для языка C# отсутствуют инструменты, способные эффективно выявлять нетривиальные случаи модификации коллекций во время перечисления. Настоящая работа предлагает детектор, основанный на символьном анализе с полной поддержкой межпроцедурности, что позволяет обнаруживать ошибки в реальных проектах, где модификация коллекции происходит в телах методов, вызываемых из цикла.

1.3 Коллекции в языке C# и их перечисление

Коллекция в языке программирования C# – абстракция, использующаяся для хранения и управления группами объектов. Для каждого типа коллекции определены методы работы с её элементами, не зависящие от конкретного типа хранимых данных. Среди всех типов коллекций существуют те, которые реализуют интерфейс `IEnumerable`, определяющий возможность последовательного перебора элементов коллекции, благодаря обязательному наличию в нём метода `GetEnumerator()`. Это предоставляет возможность перечисления (итерации по) коллекции.

Для перечисления коллекции в языке C# существует оператор `foreach`. Использование оператора `foreach` преобразуется компилятором в последовательность вызовов `GetEnumerator()`, `MoveNext()` и `Current` [11]. В табл. 1 показано сравнение использования оператора `foreach` (листинг 2) и эквивалентного кода, полученного в результате трансформации для дальнейшей компиляции (листинг 3).

При получении перечислителя (`enumerator` в листинге 3) состояние коллекции на момент начала обхода фиксируется в самом перечислителе: в структуру перечислителя сохраняется копия коллекции и номер её версии. Номер версии увеличивается при любых модификациях коллекции. При каждом вызове метода `MoveNext()` выполняется проверка соответствия между сохранённой в перечислителе версией и актуальной версией коллекции. В случае их расхождения генерируется исключение InvalidOperationException. Таким образом, описанный способ обхода коллекции не допускает её модификации в процессе перечисления.

Табл. 1. Реализация цикла foreach.
Table 1. Implementation of foreach cycle.

```
1  foreach(int elem in testList)
2  {
3      testList.Add(1);
4  }
```

Листинг 2. Простейший цикл foreach.
Listing 2. Simple foreach loop.

```
1  IEnumerator<int> enumerator =
                testList.GetEnumerator();
2  try
3  {
4      while (enumerator.MoveNext())
5      {
6          int elem =
                    enumerator.Current;
7          testList.Add(1);
8      }
9  }
10 finally
11 {
12     IDisposable disposable =
                enumerator as IDisposable;
13     if (disposable != null)
14         disposable.Dispose();
15 }
```

Листинг 3. Представление цикла foreach.
Listing 3. Foreach loop representation.

2. Описание схемы работы детектора

2.1 Общая схема работы SharpChecker

В данной статье рассматривается детектор, реализованный в промышленном статическом анализаторе языка C# – SharpChecker [12]. Анализатор разработан на базе Roslyn [13] – платформы с открытым исходным кодом, предоставляющей компиляторы и инструменты для статического анализа кода на языке C#.

Процесс анализа состоит из нескольких этапов. На первом этапе исходный код преобразуется в абстрактное синтаксическое дерево (АСД). Затем строится статический граф вызовов функций, обход которого позволяет сформировать для каждого метода резюме, содержащее необходимую для анализа информацию о методе. Во время анализа обход графа вызовов выполняется снизу-вверх – от листовых блоков графа вызовов к корневым, что позволяет сохранять в резюме каждого метода информацию как о нём самом, так и о вызываемых им методах.

На следующем этапе для каждого метода строится граф потока управления (ГПУ). Узлами этого графа являются базовые блоки, каждый из которых содержит набор последовательных инструкций. Структура ГПУ отражает все возможные пути выполнения программы. Анализ осуществляется путём распространения контекста – структуры данных, содержащей накопленную на пути выполнения программы информацию о методе. Контекст передаётся по направлению к последующим блокам и дополняется при необходимости. В точках слияния контексты объединяются.

Важным аспектом анализа является параметризация состояния каждого метода с помощью уникальных символьных значений. Это помогает представить результаты выполнения операции в виде выражений над символьными переменными, что составляет основу символьного выполнения.

2.2 Общая схема работы детектора

Ошибка модификации коллекции во время перечисления возникает, когда в теле цикла `foreach` выполняются операции, изменяющие коллекцию, которая перечисляется в цикле. К числу таких операций относятся:

- Методы, изменяющие состав коллекции (`Add`, `Remove`, `Clear` и другие).
- Операции индексированного присваивания (например, `dict[key] = value`).
- Изменение свойств, влияющих на внутреннее представление данных (например, свойства `Capacity` у коллекции `List<T>`).

Для того, чтобы обнаружить такую ошибку, разработан следующий алгоритм: детектор фиксирует получение перечислителя для коллекции (то есть вызов метода `GetEnumerator()`) – это может сигнализировать о начале цикла по этой коллекции. Далее, пока не встретится повторный вызов метода `MoveNext()`, анализируется тело цикла и происходит поиск в нём модификаций коллекции. Для анализа интересен именно второй вызов этого метода, так как первый вызов лишь инициирует цикл, перемещая перечислитель к первому элементу, и в этот момент тело цикла ещё не обработано. Таким образом, только при обнаружении второго вызова `MoveNext()` можно сделать вывод о завершении первой итерации и проверить, не содержала ли она модификаций коллекции, и, если содержала, сгенерировать соответствующие предупреждения.

2.3 Внутрипроцедурный анализ

Для обнаружения подобных модификаций используются обработчики событий (таких как вызов метода, обращение к полю элемента класса и т. д.), вызываемые из ядра символьного анализа при обходе ГПУ (пример ГПУ на уровне промежуточного представления IOperation [14], предоставленного компилятором Roslyn, для простейшего цикла из листинга 2 представлен на рис. 1). Состояние программы в каждом базовом блоке графа представлено контекстом, который объединяет информацию, полученную по всем путям выполнения, ведущим к этому блоку.

Перечисление производится с помощью перечислителя, возвращаемого вызываемым в начале цикла методом `GetEnumerator()`. Поэтому, при обработке вызова метода `GetEnumerator()` в контекст фиксируется соответствие между символьным значением возвращаемого перечислителя и символьным значением исходной коллекции.

В примере (рис. 1) этому этапу соответствует обработка блока B2, где происходит вызов метода `GetEnumerator()` для коллекции `testList`.

Далее для каждого цикла `foreach` в контексте устанавливается соответствие между символьным значением перечислителя, полученного при вызове `GetEnumerator()`, и символьным значением экземпляра перечислителя, используемого при вызове метода `MoveNext()` (этот этап соответствует моменту анализа базового блока B3 на рис. 1).

Это обусловлено тем, что в C# перечислители различных коллекций могут быть как структурами [15], так и классами [16]. Если перечислитель является структурой, то в момент присваивания результата вызова `GetEnumerator()` локальной переменной создаётся его копия. То есть при выполнении оператора `IEnumerator<int> enumerator = testList.GetEnumerator();` переменная `enumerator` получает копию структуры, возвращённой методом. Поэтому символьные значения исходного перечислителя (возвращённого `GetEnumerator()`) и его копии, используемой внутри цикла, будут различаться. В случае, когда перечислитель реализован как класс, присваивание происходит по ссылке, и все операции выполняются с одним и тем же объектом, поэтому его символьное значение не меняется.

При обнаружении вызова метода, модифицирующего коллекцию, анализатор проверяет контекст на наличие перечислителя, ассоциированного с символьным значением целевой коллекции. В случае обнаружения такой связи информация о ней (а именно локация модификации и условие достижимости модификации) сохраняется в контекст для дальнейшего анализа.

В примере (рис. 1) модификацию коллекции содержит блок B4: `Invocation Void [B4.4].Add([B4.6])`.

При обнаружении метода `MoveNext()` анализатор выполняет поиск символьного значения исходного перечислителя в контексте (это необходимо в случае, если перечислитель является структурой) и извлечение символьного значения коллекции, для которой был получен данный перечислитель.

На следующем этапе контекст проверяется на наличие информации о модификациях этой коллекции. Если такие данные отсутствуют, это означает один из двух вариантов: либо обрабатывается самый первый вызов `MoveNext()` (на рис. 1 блок B3), либо, если происходит обработка повторного вызова (на рис. 1 блок B3), модификации коллекции не были обнаружены внутри цикла по ней. В ГПУ содержится два блока с вызовом метода `MoveNext()` для удобства представления цикла в анализаторе: с помощью такого представления при обходе ГПУ есть возможность выйти из цикла без прохода по обратному ребру. Если записи о модификациях присутствуют, то для каждой из них анализатор вызывает SMT-решатель, который проверяет совместность условий достижения места изменения коллекции. Если модификация является достижимой, создаётся диагностика, после чего соответствующая информация удаляется из контекста для освобождения памяти.

2.4 Межпроцедурный анализ

Описанный метод нахождения ошибки с помощью внутрипроцедурного анализа не применим в том случае, если модификация коллекции происходит внутри тела метода, вызываемого через последовательную цепочку вызовов других методов, которая начинается в теле цикла, как в мотивационном примере (листинг 1). Для отслеживания такого сценария модификации необходим межпроцедурный анализ. Для этого используется подход на основе резюме методов. В резюме сохраняется информация обо всех обнаруженных вызовах методов, модифицирующих коллекций: символьное значение коллекции, с которой была произведена модификация, локация и условие достижимости модификации для последующего создания предупреждения при необходимости. В случае, если при анализе вызова метода выясняется, что в контексте вызывающего метода содержится информация о получении перечислителя для модифицируемой вызываемым методом коллекции, информация из сформированного резюме переносится в контекст вызывающего метода. Таким образом, в контекст сохраняется не только информация о явных изменениях коллекции (происходящих непосредственно в цикле), но и данные о скрытых модификациях (вынесенных в другие методы), выявленных в ходе межпроцедурного анализа.

2.5 Допустимые случаи модификации во время перечисления

Представленный детектор целенаправленно игнорирует некоторые сценарии модификации коллекций, которые не приводят к появлению исключений. К таким сценариям относятся:

- Работа с потокобезопасными коллекциями (System.Collections.Concurrent): перечислители коллекций из этого пространства имён специально разработаны для работы в условиях параллельных изменений, поэтому их модификация во время перечисления является допустимой операцией и не приводит к исключению InvalidOperationException [17].

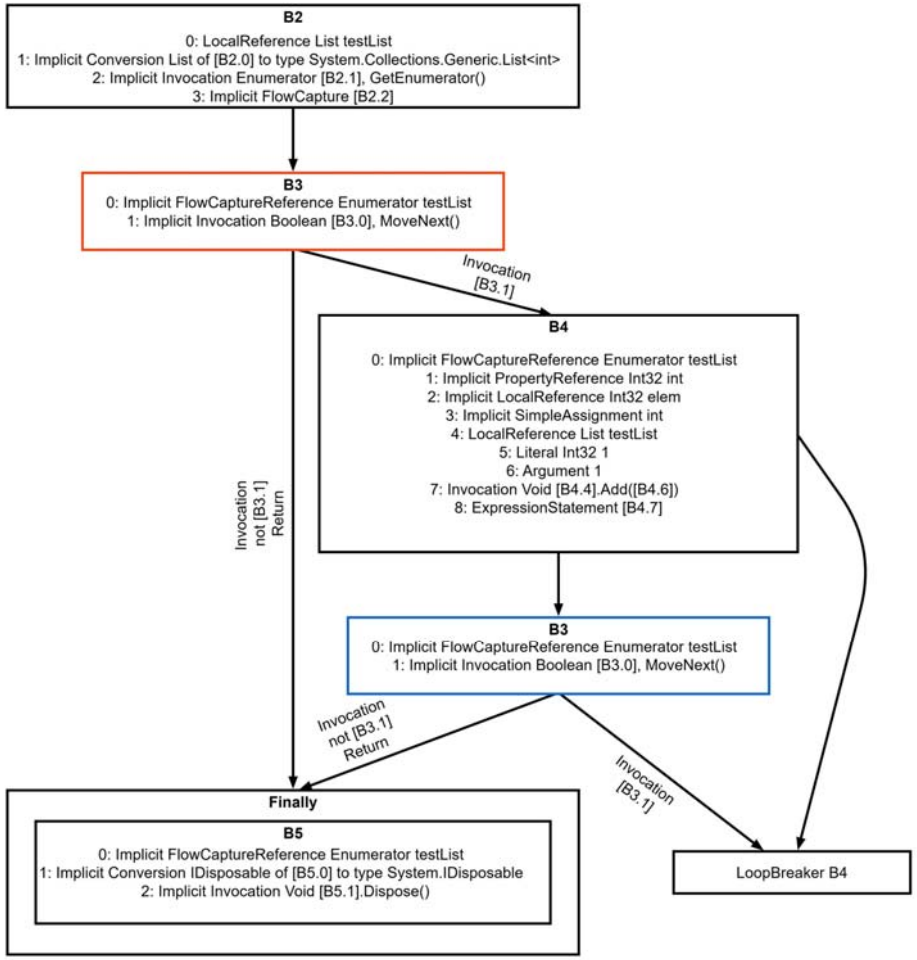

Рис. 1. Визуализация графа потока управления для листинга 2.
Fig. 1. Control flow graph visualization for listing 2.

- Немедленное завершение цикла после модификации: если за изменением коллекции внутри цикла `foreach` следует оператор `break` или выход из метода (`return`), диагностика не создаётся. Анализ графа потока управления для такого кода показывает, что после операции модификации отсутствуют пути выполнения, ведущие к последующему вызову метода `MoveNext()`. Поскольку перечислитель не будет использован после изменения коллекции, поведение программы является корректным.

3. Результаты

Тестирование проводилось на наборе из 21 проекта с открытым исходным кодом, общий объём которого составил 6 млн. строк кода. Анализ выполнялся на машине с 20-ядерным процессором Intel(R) Core(TM) i7-14700 , имеющей 64 GB RAM.

Было сгенерировано 13 предупреждений. В результате их анализа было выяснено, что 9 из них являются истинными предупреждениями, 4 – ложными, то есть доля истинных предупреждений составляет около 69%. При включении детектора модификаций коллекций в процессе перечисления время работы SharpChecker возрастает незначительно (приблизительно на 1%).

Для оценки полноты предложенного подхода проведено сравнение результатов с существующими инструментами. Все испытуемые анализаторы обнаружили две тривиальных ошибки модификации коллекций на проекте OpenSim [18]. Однако, предложенный метод показал существенное преимущество в анализе сложных сценариев, выявив три дополнительные ошибки, возникающие в результате модификации коллекции через цепочки вызовов методов. Эти случаи пропущены другими инструментами, что подтверждает необходимость использования межпроцедурного анализа и обосновывает практическую ценность разработанного подхода.

3.1 Пример 1

Рассмотрим ошибку (листинг 4), найденную на проекте OpenSim в TreePopulatorModule.cs на строках 664-686.

```
1  foreach (UUID tree in copse.m_trees) // Foreach entry
2  {
3      if (...)
4      {
5          ...
6          SpawnChild(copse, s_tree); // INVALIDATED_ITERATOR: The object
                   copse.m_trees being iterated over was modified inside
                    the loop by the expression SpawnChild(copse, s_tree)
7      }
8      ...
9  }
```

Листинг 4. Место предупреждения.
Listing 4. Location of the error.

В примере на листинге 5 коллекция `copse.m_trees` модифицируется через последовательные вызовы двух функций: `SpawnChild` и `CreateTree`. Объект класса `Copse`, который содержит в себе список `m_trees`, передается в качестве аргумента в каждую из функций. Это гарантирует, что изменения применяются именно к той коллекции, итерация по которой выполняется в данный момент. Данный случай наглядно демонстрирует необходимость и эффективность применения межпроцедурного анализа в предложенном детекторе.

3.2 Пример 2

Наиболее распространённым сценарием реальной ошибки является случай прямого удаления элементов в цикле, рассмотренный в листинге 6. Подобная ошибка была обнаружена в нескольких проектах, например, в проекте Tizen 5.5 в файле SmartcardSession.cs на строках 145-149.

В данном фрагменте выполняется попытка очистки списка `_logicalChannelList`, реализованная некорректно. Для безопасного удаления элементов в процессе итерации следует создать копию коллекции, например, с помощью метода `ToList()`, и выполнять цикл уже по этой копии. Это позволяет избежать модификации исходной коллекции во время работы перечислителя.


```
1  private void SpawnChild(Copse copse, SceneObjectPart s_tree)
2  {
3      ...
4      if (...)
5      {
6          ...
7          CreateTree(uuid, copse, position); // Collection was changed
                    in the expression CreateTree(uuid, copse, position)
8      }
9  }
10
11 private void CreateTree(UUID uuid, Copse copse, Vector3 position)
12 {   ...
14     if (...)
15     {
16         ...
17         copse.m_trees.Add(tree.UUID); // Collection was changed in the
                           expression copse.m_trees.Add(tree.UUID)
18         ...
19     }
20 }
```

Листинг 5. Реализаця методов.
Listing 5. Methods implementation.

```
1  foreach (SmartcardChannel channel in _logicalChannelList) // Foreach
                                                                 entry
2  {
3      channel.Dispose();
4      _logicalChannelList.Remove(channel); // INVALIDATED_ITERATOR: The
       object _logicalChannelList being iterated over was modified inside
       the loop by the expression _logicalChannelList.Remove(channel)
5  }
```

Листинг 6. Пример ошибки модификации коллекции во время итерации по ней.
Listing 6. The warning example of collection modification during its iteration.

3.3 Пример 3 (ложное предупреждение)

Одно из ложных предупреждений было сгенерировано при анализе кода проекта ShareX в файле TaskSettingsForm.cs на строках 249-25 (листинги 7 и 8). В листинге 8 (строка 14) перед добавлением элемента выполняется проверка его наличия в коллекции. Поскольку добавляемый элемент уже содержится в коллекции, операция модификации не происходит.

```
1  foreach (WatchFolderSettings watchFolder in
                              TaskSettings.WatchFolderList) // Foreach entry
2  {
3      AddWatchFolder(watchFolder); // INVALIDATED_ITERATOR: The object
       TaskSettings.WatchFolderList being iterated over was modified inside
                    the loop by the expression AddWatchFolder(watchFolder)
4  }
```

Листинг 7. Место предупреждения.
Listing 7. Location of the error.

Ложное предупреждение возникает вследствие ограниченной поддержки анализатором моделирования коллекций. В частности, в представленном на листинге случае, недостаточно точно моделируется состояние коллекции, а также отсутствует межпроцедурный анализ содержимого коллекций. То есть анализатор не позволяет точно определить, содержится ли

элемент в коллекции. В дальнейшем планируется доработать механизм анализа содержимого коллекций, что позволит устранить подобные ложные предупреждения.

```
1  private void AddWatchFolder(WatchFolderSettings watchFolderSetting)
2  {
3      if (...)
4      {
5          Program.WatchFolderManager.AddWatchFolder(watchFolderSetting,
                                                      TaskSettings); //
                                    Collection was changed in the expression
           Program.WatchFolderManager.AddWatchFolder(watchFolderSetting,
                                                                TaskSettings)
6          ...
7      }
8  }
9
10 public void AddWatchFolder(WatchFolderSettings watchFolderSetting,
                                         TaskSettings taskSettings)
11 {
12     if (...)
13     {
14         if(!taskSettings.WatchFolderList.Contains(watchFolderSetting))
15         {
16             taskSettings.WatchFolderList.Add(watchFolderSetting); //
                                  Collection was changed in the expression
                        taskSettings.WatchFolderList.Add(watchFolderSetting)
17         }
18         ...
19     }
20 }
```

Листинг 8. Реализация методов.
Listing 8. Methods implementation.

4. Заключение

Ошибки модификации коллекций в процессе перечисления регулярно возникают при разработке на языке C#, что приводит к аварийному завершению программы из-за исключения InvalidOperationException. Они присутствуют даже в протестированном коде, проявляясь в редких сценариях использования. В связи с этим необходима их поддержка в статических анализаторах. Существующие решения обладают ограниченной полнотой анализа и способны обнаруживать лишь простые случаи модификации, не охватывая сложные сценарии с вызовом методов.

Реализованный алгоритм на основе межпроцедурного, чувствительного к потоку управления и путям выполнения программы механизма символьного выполнения промышленного статического анализатора SharpChecker продемонстрировал высокую полноту и допустимую точность при тестировании на реальных проектах.

Экспериментальная оценка на 21 проекте с открытым исходным кодом подтвердила практическую ценность решения: доля истинных предупреждений составила около 69%. Такие ошибки действительно допускаются в реальных проектах, что доказывает актуальность создания данного детектора, способного обнаруживать их с приемлемой точностью. Реализация разработанного подхода незначительно влияет на время анализа (в пределах 1%) и внедрена в промышленный анализатор SharpChecker.

Список литературы / References

[1]. Tiobe index for ranking the popularity of programming languages. Available at: https://www.tiobe.com/tiobe-index, accessed 12.03.2026.

[2]. Bodden E., Lam P., Hendren L. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. 2008. pp. 36-47.

[3]. Horváth G., Páter-Részeg A., Pataki N. Detecting Misusages of the C++ Standard Template Library. Proceedings of the 10th International Conference on Applied Informatics (ICAI 2017), 2017, pp. 129-136.

[4]. Clang: a C language family frontend for LLVM. Available at: https://clang.llvm.org, accessed 12.03.2026.

[5]. GDB: The GNU Project Debugger. Available at: https://www.sourceware.org/gdb/, accessed 12.03.2026.

[6]. Иванников В.П., Белеванцев А.А. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 231-250. DOI: 10.15514/ISPRAS-2014-26(1)-7 / Ivannikov V.P., Belevantsev A.A. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, issue 5, 2014, pp. 265-275.

[7]. PVS-Studio: Static Code Analyzer. Available at: https://pvs-studio.ru/ru/, accessed 12.03.2026.

[8]. V3221: Modifying a collection during its enumeration will lead to an exception. PVS-Studio Documentation. Available at: https://pvs-studio.ru/ru/docs/warnings/v3221/, accessed 12.03.2026.

[9]. ReSharper: The Visual Studio Extension for .NET Developers. Available at: https://www.jetbrains.com/resharper/, accessed 12.03.2026.

[10]. Possible 'System.InvalidOperationException: Collection was modified'. Code Inspections in C#. ReSharper Documentation. Available at (accessed 12.03.2026): https://www.jetbrains.com/help/resharper/Code_Analysis__Code_Inspections.html.

[11]. IEnumerable.GetEnumerator Метод. Available at: https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.ienumerable.getenumerator?view=net-8.0, accessed 12.03.2026.

[12]. Кошелев В.К., Игнатьев В.Н., Борзилов А.И. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 21-40. DOI: 10.15514/ISPRAS-2016-28(1)-2 / Koshelev V.K., Ignatiev V.N. et al. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, vol. 43, issue 4, 2017, pp. 268-276.

[13]. dotnet/roslyn: The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs. Available at: https://github.com/dotnet/roslyn, accessed 12.03.2026.

[14]. ControlFlowGraph Class. Available at: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.flowanalysis.controlflowgraph?view=roslyn-dotnet-5.0.0, accessed 12.03.2026.

[15]. List<T> Source Code. .NET Foundation. Available at: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Collections/Generic/List.cs, accessed 12.03.2026.

[16]. ConcurrentDictionary Source Code. .NET Foundation. Available at: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Collections.Concurrent/src/System/Collections/Concurrent/ConcurrentDictionary.cs, accessed 12.03.2026.

[17]. ConcurrentDictionary<TKey,TValue>.GetEnumerator Метод. Available at: https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentdictionary-2.getenumerator?view=net-8.0, accessed 12.03.2026.

[18]. OpenSimulator is an open source multi-platform, multi-user 3D application server. Available at: http://opensimulator.org/wiki/Main_Page, accessed 12.03.2026.

Информация об авторах / Information about authors

Ксения Андреевна ХАНЕВСКАЯ – студентка бакалавриата Физтех-школы Радиотехники и Компьютерных Технологий Московского физико-технического института, сотрудник ИСП РАН. Научные интересы: статический анализ программ, статическое символьное выполнение, алгоритмы поиска дефектов в исходном коде.

Ksenia Andreevna KHANEVSKAYA – bachelor's student at the Department of Radio Engineering and Computer Technologies of the Moscow Institute of Physics and Technology, an employee of

the ISP RAS. Research interests: static program analysis, static symbolic execution, algorithms for finding defects in source.

Ульяна Владимировна ТЯЖКОРОБ – студентка аспирантуры Физтех-школы Радиотехники и Компьютерных Технологий Московского физико-технического института, сотрудник ИСП РАН. Научные интересы: статический анализ программ, статическое символьное выполнение, алгоритмы поиска дефектов в исходном коде.

Uljana Vladimirovna TSIAZHKOROB is a postgraduate student at the Department of Radio Engineering and Computer Technologies of the Moscow Institute of Physics and Technology, an employee of the ISP RAS. Research interests: static program analysis, static symbolic execution, algorithms for finding defects in source.

Валерий Николаевич ИГНАТЬЕВ, кандидат физико-математических наук, старший научный сотрудник ИСП РАН, доцент кафедры системного программирования факультета ВМК МГУ. Научные интересы включают методы поиска ошибок в исходном коде ПО на основе статического анализа.

Valery Nikolayevich IGNATYEV – Cand. Sci. (Phys.-Math.) in computer sciences, senior researcher at Ivannikov Institute for System Programming RAS and associate professor at system programming division of CMC faculty of Lomonosov Moscow State University. He is interested in techniques of errors and vulnerabilities detection in program source code using static analysis.