DOI: 10.15514/ISPRAS-2025-37(6)-18

Статический анализ языка Visual Basic .NET

[1,2] *Карцев В. С., ORCID: 0000-0001-7482-0835 <karcev.vs@ispras.ru>*
[1,3] *Игнатьев В. Н., 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] *Физтех-школа радиотехники и компьютерных технологий МФТИ, 141701, Россия, г. Долгопрудный, ул. Первомайская, д. 5.*

[3] *Факультет вычислительной математики и кибернетики МГУ им. М.В. Ломоносова, 119991, Россия, г. Москва, ул. Колмогорова, д. 1, стр. 52.*

Аннотация. В работе представлена реализация статического анализа для языка Visual Basic .NET в рамках промышленного инструмента SharpChecker. С помощью фреймворка компилятора Roslyn в SharpChecker была интегрирована поддержка языка Visual Basic .NET. Это позволило выполнять статический анализ исходного кода на языке Visual Basic .NET. В рамках работы также был создан репрезентативный набор синтетических тестов, содержащий суммарно более 2000 тестов. Тестирование производилось как на созданной выборке тестов, так и на наборе реальных проектов с открытым исходным кодом суммарным объемом более 1.6 млн. строк кода. Было обнаружено 7926 новых предупреждений в исходном коде на языке Visual Basic .NET, из которых 1093 были проанализированы и размечены вручную. Итоговая точность анализа составила 84.72%. Кроме того, были обнаружены предупреждения, связанные с кодом на языках C# и Visual Basic .NET одновременно, что показало возможность производить межъязыковой анализ в проектах, которые содержат сразу два языка платформы .NET. Добавление поддержки языка Visual Basic .NET в инструмент SharpChecker не отразилось на времени работы и на качестве анализа для языка C#.

Ключевые слова: статический анализ; Visual Basic .NET; символьное исполнение; анализ потоков данных; генерация кода.

Для цитирования: Карцев В. С., Игнатьев В. Н. Статический анализ языка Visual Basic .NET. Труды ИСП РАН, том 37, вып. 6, часть 2, 2025 г., стр. 37–52. DOI: 10.15514/ISPRAS–2025–37(6)–18.

Static Analysis of Visual Basic .NET Language

[1,2] *Karcev V. S., ORCID: 0000-0001-7482-0835 <karcev.vs@ispras.ru>*
[1,3] *Ignatiev V. N., ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>*

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

[2] *Departament of Radio Engineering and Cybernetics MIPT, 5, Pervomayskaya Str., Dolgoprudny, 141701, Russia.*

[3] *CMC Faculty Lomonosov State University, 1c52, Kolmogorov Str., Moscow, 119991, Russia.*

Abstract. The paper presents the implementation of static analysis for the Visual Basic .NET language within the industrial tool SharpChecker. Using the Roslyn compiler framework, support for the Visual Basic .NET language was integrated into SharpChecker, enabling static analysis of Visual Basic .NET source code. As part of this work, a representative set of synthetic tests was created, comprising over 2000 test cases. Testing was conducted both on this synthetic dataset and on a collection of real-world open-source projects totaling more than 1.6 million lines of code. A total of 7926 new warnings were detected in Visual Basic .NET source code, of which 1093 were manually reviewed and labeled. The final analysis accuracy reached 84.72%. Additionally, warnings related to code written in both C# and Visual Basic .NET were discovered, demonstrating the feasibility of cross-language analysis in projects that include both .NET platform languages. It was also found that adding Visual Basic .NET language support to SharpChecker had no impact on the performance or the quality of analysis for the C# language.

Keywords: static analysis; Visual Basic .NET; symbolic execution; dataflow analysis; code generation

For citation: Karcev V. S., Ignatiev V. N. Static analysis of Visual Basic .NET language. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 2, 2025, pp. 37-52 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-18.

1. Введение

Visual Basic .NET – это интерпретируемый, объектно-ориентированный язык со статической типизацией, разработанный корпорацией Microsoft. Visual Basic .NET произошёл от языка Visual Basic (сейчас известен как Visual Basic 6.0 или VB6), также разработанного компанией Microsoft. Несмотря на то, что Visual Basic .NET – это очередное, хоть и значительное, обновление языка Visual Basic, обратная совместимость с VB6 сохранена не была [1]. Ключевой особенностью языка является его ориентированность на среду исполнения .NET при высокой простоте перехода с VB6. Для упрощения перехода на платформу .NET компанией Microsoft была разработана утилита Migration Wizard, которая позволяет конвертировать проекты с VB6 на Visual Basic .NET.

Visual Basic .NET является восьмым по популярности языком согласно рейтингу TIOBE [2]. Он широко используется в корпорациях и государственных организациях, что объясняется существованием большой устоявшейся кодовой базы на Visual Basic .NET и VB6 внутри компаний. При этом, переход с устаревшего VB6 на Visual Basic .NET может повлечь большое количество ошибок в существующих проектах, так как утилита Migration Wizard не гарантирует корректности и конвертированный код требует ручной доработки [3].

Также стоит отметить, что в некоторых современных проектах, ориентированных на платформу .NET прослеживается тенденция на использование одновременно нескольких языков платформы .NET. В таких проектах игнорирование исходного кода на языке Visual Basic .NET может привести к пропуску ошибок, связанных с этим кодом. В качестве примера такого проекта можно рассмотреть компилятор для языков C# и Visual Basic .NET Roslyn [4], включающий 1.52 млн. строк кода на Visual Basic .NET. Для того чтобы считать такое ПО безопасным, согласно ГОСТ Р 56939-2024 о разработке безопасного ПО, необходимо производить, в том числе, контроль качества исходного кода [5]. То есть, для соблюдения

требований ГОСТ, необходимо производить статический анализ не только C#, но и Visual Basic .NET.

Для оценки сложности интеграции и возможного качества результатов авторами ранее был реализован прототип поддержки языка Visual Basic .NET в промышленном инструменте SharpChecker. Прототипирование показало, что без кардинальных изменений анализатора можно достигнуть сравнимых с C# показателей полноты и точности [6]. При этом появилась возможность производить статический анализ всех типов – как синтаксический анализ, так и более сложные, чувствительные к контексту и путям исполнения типы анализа. Это показало актуальность доработки прототипа до полноценного решения, которое может быть предоставлено конечным пользователям.

Статический анализ исходного кода на языке Visual Basic .NET был реализован в рамках промышленного статического анализатора SharpChecker, разработанного в ИСП РАН [7]. SharpChecker предусматривает следующий конвейер анализа:

1. перехват сборки и сбор информации для индексации исходного кода;
2. повторная сборка и анализ, разделенный на следующие этапы:
   a. синтаксический анализ;
   b. анализ потоков данных;
   c. чувствительный к путям выполнения межпроцедурный анализ на основе символьного выполнения;
   d. межпроцедурный, контекстно-чувствительный анализ помеченных данных [8];
3. отображение обнаруженных предупреждений в графическом пользовательском интерфейсе с возможностью разметки и навигации по исходному коду.

Для проверки работоспособности и отладки инструмента SharpChecker используются синтетические тесты, представляющие собой корректные компилируемые программы с допущенными и размеченными ошибками.

Помимо анализа исходного кода, SharpChecker собирает метрики исходного кода – численные значения, отражающие его характеристики. В качестве примера метрик можно привести цикломатическую сложность, связность и другие [9]. Данные метрики могут быть использованы для дальнейшего анализа качества кода или автоматической разметки сгенерированных предупреждений [10].

Архитектура SharpChecker подразумевает наличие отдельных модулей (далее детекторов), производящих поиск определенных типов дефектов или производящих сбор дополнительных данных, например, информации о значениях полей [11].

SharpChecker был выбран в качестве платформы для реализации статического анализа Visual Basic .NET, так как он использует для сборки и построения промежуточных представлений исходного кода компилятор для C# и Visual Basic .NET Roslyn [4]. Это позволяет производить межъязыковой анализ для проектов, использующих оба эти языка.

Целью данной работы является разработка методов и алгоритмов для поддержки статического анализа кода на Visual Basic .NET в инструменте SharpChecker.

Для достижения поставленной цели необходимо:

- разработать механизмы поддержки языка Visual Basic .NET на всех этапах анализа;
- создать репрезентативный набор тестов;
- обеспечить навигацию по исходному коду в интерфейсе пользователя;
- разработать подсистему сбора метрик исходного кода для языка Visual Basic .NET;
- произвести тестирование реализованных методов и механизмов на наборе проектов с открытым исходным кодом.

2. Существующие решения

Большинство промышленных статических анализаторов не поддерживают язык Visual Basic .NET [12]. Например, Klocwork [13] и PVS-Studio не поддерживают Visual Basic .NET, несмотря на то, что они поддерживают C# и платформу .NET.

Тем не менее, некоторые статические анализаторы реализуют анализ языка Visual Basic .NET. Например, Visual Basic .NET поддерживается в инструменте ReSharper [14-15]. ReSharper направлен в первую очередь на быстрый анализ кода на языках C# и Visual Basic .NET для обеспечения помощи разработчику непосредственно при написании кода. Поиск большинства типов ошибок в ReSharper производится в рамках одного файла. В рамках всего проекта возможен только поиск ошибок, связанных с использованием и доступом к членам объявленных типов. Таким образом, при анализе ReSharper не учитывает содержимое других файлов, что приводит к невозможности обнаружить нетривиальные ошибки, для поиска которых необходим трудоемкий межпроцедурный и межмодульный анализ.

Статический анализ языка Visual Basic .NET также представлен в инструменте SonarQube [16-17]. Этот инструмент используется для обнаружения ошибок, уязвимостей и подозрительных мест в коде на основе правил. Он поддерживает 21 язык, среди которых присутствует и Visual Basic .NET. Отметим, что количество правил для языка Visual Basic .NET значительно меньше, чем для других языков. Большинство правил направлены на поиск простых ошибок, таких как поиск классов, объявленных вне пространства имен или использование слабых криптографических алгоритмов. Однако существуют и более сложные правила, например, отсутствие снятия блокировки на одном или нескольких путях исполнения или разыменование `Nothing` [18].

Инструмент поиска уязвимостей Kiuwan также поддерживает анализ языка Visual Basic .NET. Kiuwan предназначен для обнаружения уязвимостей и оценки качества исходного кода. Kiuwan обнаруживает, в том числе, переполнения буфера, инъекции команд, межсайтовый скриптинг, SQL-инъекции и другие типы уязвимостей [19]. Однако данный инструмент не предназначен для обнаружения других типов ошибок, таких, как недостижимый код или разыменование `null` (или `Nothing` в случае Visual Basic .NET).

3. Интеграция языка Visual Basic .NET

3.1 Генерация тестовой выборки

Для отладки и проверки инструмента SharpChecker в процессе добавления поддержки Visual Basic .NET необходим набор тестов. В SharpChecker уже существует репрезентативный набор тестов, созданный более, чем за 5 лет для языка C#. По этой причине было принято решение создать инструмент автоматической трансляции набора тестов с C# на Visual Basic .NET для создания аналогичного набора тестов для нового языка с минимальными затратами по времени. В рамках статьи [6] был описан инструмент автоматической генерации набора синтетических тестов. Данный инструмент состоит из двух компонентов: механизма импорта и экспорта тестовой выборки и механизма трансляции кода с языка C# на язык Visual Basic .NET.

Для импорта всех тестов в инструмент трансляции в первую очередь необходимо обнаружить все файлы, содержащие тесты. Далее для каждого файла, в котором были обнаружены тесты, с помощью Roslyn создается абстрактное синтаксическое дерево и производится поиск функций, помеченных атрибутом `[Test]`. В таких функциях исходный код программы с размеченными ошибками передается в функцию `VerifyCSharp`, производящую анализ этого исходного кода и сверяющего результаты анализа с ручной разметкой. Данный исходный код извлекается из вызова проверочной функции и передается на трансляцию из C# на Visual Basic .NET.

В первую очередь, перед трансляцией производится переименование символов, объявленных в коде, с помощью их нумерации. Далее для исходного кода строится синтаксическое дерево и создается эквивалентное из аналогичных узлов для Visual Basic .NET. После перевода всех тестов транслятор воссоздает структуру исходного синтаксического дерева файла с тестами, заменяя проверочные методы для C# на аналогичные для Visual Basic .NET и подставляя в качестве параметра переведенный исходный код.

Из 2680 тестов для языка C#, содержащихся в SharpChecker, на язык Visual Basic .NET удалось автоматически перевести 1928 тестов. Тесты, которые не удалось перевести, используют функциональность C#, которая не имеет аналогий в Visual Basic .NET. Таким образом, в результате использования инструмента автоматического перевода тестов удалось получить репрезентативный набор тестов для языка Visual Basic .NET, практически эквивалентный аналогичному для языка C#.

Заметим, что данный инструмент производит перевод на основе синтаксического дерева, заменяя конструкции языка C# соответствующими для языка Visual Basic .NET. Благодаря этому, переведенный исходный код после преобразования во внутреннее представление компилятора Roslyn, будет трактоваться эквивалентно аналогичному на языке C#. Таким образом, данный метод позволяет гарантировать отсутствие различий в семантике исходного кода.

3.2 Адаптация существующих детекторов

Анализ исходного кода в инструменте SharpChecker происходит с использованием различных методов анализа. В первую очередь стоит отметить детекторы, производящие синтаксический анализ. На данном этапе, помимо абстрактного синтаксического дерева, доступны также таблица символов и дерево операций [20]. Операции, предоставляемые платформой Roslyn, являются универсальным представлением различных паттернов исходного кода, общих для всех языков. Так, например, существуют операции, обозначающие в обобщенном виде циклы, ветвления, методы, классы и так далее. Однако существует также множество специфичных для языка конструкций, которые не имеют аналогов в виде операций.

В основном синтаксический анализ использует обработку либо узлов АСД, либо символов, либо операций, либо их комбинации. В случае, если детектор использует только операции и/или символы, то его доработка необходима, только если существует некая специфика, присущая языку Visual Basic .NET. В остальных случаях такие детекторы способны обрабатывать код на любом из языков. В случае, если детектор использует обработку синтаксических узлов (например, если аналогичных им операций не существует), необходимо добавить поддержку аналогичных синтаксических узлов для Visual Basic .NET.

После синтаксического анализа следует этап символьного исполнения. Детекторы на этапе символьного исполнения имеют сложный граф зависимостей. Часть детекторов используется для предварительного анализа исходного кода и сбора данных о нем. Назовем набор таких детекторов «ядром». Обычные детекторы, в свою очередь, используют собранную им информацию. Таким образом, в первую очередь необходимо реализовать поддержку языка Visual Basic .NET именно в ядре. Это поможет инструменту SharpChecker собирать информацию об исходном коде на языке Visual Basic .NET и составлять резюме методов для дальнейшего анализа. Для устранения ошибок и неточностей в анализе отдельные детекторы также требуют доработки.

Детекторы, использующие анализ помеченных данных, работают несколько иначе. Каждый детектор на данном этапе представлен отдельным правилом, описанным в виде JSON-файла. Движок анализа помеченных данных использует эти правила для определения характеристик стока и истока помеченных данных, а также пути распространения этих данных. Таким образом, сами правила не нуждаются в доработке, за исключением добавления новых стоков,

истоков или распространителей, специфичных только для языка Visual Basic .NET. Доработка необходима только основному движку.

По большей части, некорректная работа инструмента SharpChecker с исходным кодом на языке Visual Basic .NET связана с тем, что Visual Basic .NET и C# имеют разные типы синтаксических узлов. В результате детекторы, реализующие анализ узлов АСД, игнорируют узлы АСД для кода, написанного на Visual Basic .NET. Помимо этого, язык Visual Basic .NET имеет ряд отличий от языка C#:

- встроенный XML-синтаксис;
- инициализация полей вызовом конструктора с помощью оператора `As`;
- сравнение объектов по ссылке с помощью отдельных операторов `Is` и `IsNot`;
- дополнительный метод `IsNothing` для сравнения объектов ссылочного типа со значением `Nothing`;
- оператор `ReDim` для реаллокации массива;
- иное представление цепочки ветвлений в АСД (вся цепочка в виде списка вместо вложенных узлов);

3.2.1 Синтаксический анализ

Основными особенностями детекторов, производящих синтаксический анализ, являются высокая скорость анализа, низкое потребление ресурсов и независимость от других компонентов инструмента SharpChecker. Это приводит к необходимости производить доработку каждого детектора по отдельности до достижения приемлемого качества анализа.

Помимо синтаксического дерева, Roslyn предоставляет универсальные представления исходного кода, такие как дерево операций или таблица символов. В инструменте SharpChecker присутствуют детекторы, которые изначально были реализованы с использованием только анализа дерева операций или таблицы символов. К таким детекторам можно отнести, например, `UselessCall` или `DuplicateEnumMember`. Однако большинство детекторов используют анализ узлов АСД, что приводит к необходимости их доработки. Для различных детекторов были использованы различные подходы к адаптации:

- поиск неиспользуемых полей и забытых модификаторов `ReadOnly` – добавлена поддержка XML-синтаксиса, добавлена поддержка синтаксических узлов Visual Basic .NET;
- детектор пустых интерфейсов и детектор идентичных методов – добавлена обработка синтаксических узлов Visual Basic .NET;
- поиск перекрытия имен символов – анализ узлов АСД был заменен анализом операций и символов;
- обнаружение скопированных участков кода с некорректными изменениями – добавлен анализ узлов АСД для Visual Basic .NET при анализе компиляции;
- поиск некорректных сравнений чисел с плавающей точкой и сравнений чисел с плавающей точкой с целыми, поиск доступа к переменным с некорректными блокировками на основе статистики, сравнений объектов не ссылочного типа по ссылке, детектор использования слабых криптографических методов, поиск сравнений объектов не ссылочного типа с `null` (`Nothing`), поиск некорректных использований атрибута `[ThreadStatic]`, неверных использований форматных строк, присваиваний переменных или полей самих в себя, идентичных возвращаемых значений на разных путях исполнения, детектор отсутствия вызова базового метода в наследнике, поиск явно заданных математических констант и конструкций `switch - case`, не рассматривающих все члены перечисления – переписаны с использованием операций.

3.2.2 Анализ потоков данных

На этапе анализа потоков данных реализован только один детектор – `UnusedValue`. Он позволяет обнаруживать значения, не использованные до момента выхода из области видимости или до замещения другим значением.

При анализе он использует только обработку операций, что позволяет ему обнаруживать ошибки без значительных доработок. Однако, он не поддерживает обработку встроенного в Visual Basic .NET синтаксиса XML. Таким образом, для удаления ложных предупреждений о неиспользуемых значениях необходимо добавить поддержку данного вида синтаксиса. Кроме того, в Visual Basic .NET невозможно производить перехват исключений определенного типа без присвоения их в переменную. Таким образом, предупреждения о неиспользованных значениях перехваченных исключений являются заведомо ложными, и их необходимо подавлять при анализе.

3.2.3 Символьное исполнение

На этапе символьного исполнения производится межпроцедурный, чувствительный к путям и контексту выполнения анализ, позволяющий обнаруживать сложные ошибки [21]. Из-за использования эвристик, сокращающих время анализа сложных конструкций без значительных потерь в качестве анализа, этот тип анализа допускает ложные предупреждения и пропущенные ошибки [22].

Для реализации поддержки анализа языка Visual Basic .NET необходимо в первую очередь адаптировать ядро. Для этого следует добавить обработку узлов АСД для языка Visual Basic .NET, а также реализовать анализ специфичной для Visual Basic .NET функциональности. В частности, в рамках ядра необходимо анализировать XML-литералы, инициализации с помощью оператора As, использования специфичных для Visual Basic .NET методов, таких как `IsNothing` и использования операторов `ReDim`.

Для некоторых детекторов на данном этапе требуются доработки для подавления ложных предупреждений. В частности, в отдельных детекторах производится анализ синтаксического дерева. Например, в связи с отличием структуры дерева, был доработан поиск операторов `Return` внутри ветвления в детекторе недостижимого кода для кода на языке Visual Basic .NET.

3.3 Генерация дополнительных артефактов

3.3.1 Индексация исходного кода

Индексация исходного кода – процесс сбора информации о расположениях всех объявлений, определений и использований символов, необходимый для реализации навигации по исходному коду. В рамках SharpChecker индексатор исходного кода анализирует абстрактное синтаксическое дерево проекта, собирая информацию об объявлениях, определениях и использованиях символов в исходном коде. Для индексации по коду на языке Visual Basic .NET необходимо добавить обработку специфичных для Visual Basic .NET узлов АСД. Для индексации исходного кода производится синтаксический анализ, в рамках которого обрабатываются узлы, соответствующие объявлениям, определениям и использованиям символов.

Для поддержки индексации исходного кода на языке Visual Basic .NET был реализован синтаксический анализатор, аналогичный существующему для C#. В нем были реализованы методы обработки синтаксических узлов Visual Basic .NET. При анализе проекта инструмент SharpChecker выбирает подходящий индексатор исходного кода, исходя из языка, на котором реализован анализируемый код. Кроме того, для проектов, использующих сразу оба языка

(Visual Basic .NET и C#), поддерживается индексация каждого из языков по отдельности соответствующим индексатором с последующим объединением собранных данных.

Индексатор исходного кода экспортирует данные в едином формате для всех языков. Механизм навигации по исходному коду в пользовательском интерфейсе не нуждается в доработке, так как для навигации не требуется дополнительная информация, помимо собранной индексатором.

3.3.2 Сбор метрик исходного кода

Из-за высокой схожести языков для Visual Basic .NET актуален тот же набор метрик, что и для C#. Таким образом, для получения метрик исходного кода необходимо доработать существующий в SharpChecker механизм сбора метрик для обработки нового языка. Так как сбор метрик, как и индексация исходного кода, использует анализ АСД, необходимо добавить поддержку синтаксических узлов языка Visual Basic .NET.

Механизм сбора метрик реализован на базе механизма `CSharpSyntaxWalker`. Однако в данном случае реализация дублирующего сборщика метрик для Visual Basic .NET оказалась невозможна, так как для сбора метрик необходимо иметь полный граф сущностей для всего анализируемого проекта. В связи с этим возникла необходимость производить сбор метрик в рамках одного механизма, чтобы избежать в дальнейшем сложного слияния данных из двух сборщиков.

Для реализации универсального инструмента сбора метрик был использован механизм `SyntaxWalker` платформы Roslyn. Данный механизм является базовым для `CSharpSyntaxWalker` и `VisualBasicSyntaxWalker`. В отличие от них, он не реализует методов для обработки различных типов узлов и имеет лишь три метода – `Visit` для обработки любого узла синтаксического дерева, `VisitTrivia` для обработки конструкций, не принимающих участия в компиляции исходного кода и `VisitToken` для обработки токенов, на которые был разбит исходный код при построении АСД.

Универсальный механизм сбора метрик переопределяет метод `Visit`, в котором вручную проверяется тип обрабатываемого узла, и, в зависимости от него, выбирается обработчик для этого типа узлов. Большинство узлов в языках Visual Basic .NET и C# обрабатываются идентичным образом, однако в некоторых случаях имеются значительные различия между этими языками. В частности, значительно отличается обработка ветвлений с множеством веток, что обусловлено отсутствием вложенности узлов.

4. Тестирование

Тестирование статического анализа для языка Visual Basic .NET производилось несколькими путями – на сгенерированной тестовой выборке и на наборе проектов с открытым исходным кодом. Кроме того, на основе собранных для Visual Basic .NET метрик исходного кода было произведено предсказание истинности полученных предупреждений.

4.1 Результаты на наборе синтетических тестов

Набор синтетических тестов для языка C# был собран более чем за пять лет на основе истинных и ложных ошибок, обнаруженных в реальных проектах. Таким образом, данный набор покрывает множество известных сценариев возникновения ошибок в коде. Проверка работоспособности статического анализатора на наборе переведенных на Visual Basic .NET тестов может показать приблизительное качество анализа и оценить, насколько оно уступает качеству для языка C#.

Тестовая выборка включает в себя как автоматически переведенные тесты, основанные на аналогичной тестовой выборке для языка C#, так и новые тесты, реализованные для проверки специфичных для Visual Basic .NET ситуаций. Общая тестовая выборка содержит 2038 тестов

и покрывает все поддерживаемые для Visual Basic .NET типы ошибок. Из 2038 тестов 311 тестов были исключены, так как детекторы, проверяемые этими тестами, заведомо не поддерживают язык Visual Basic .NET или же логика теста не поддерживается инструментом SharpChecker. Результаты для поддерживаемых тестов приведены в табл. 1.

Табл. 1. Результаты по отдельным типам анализа.
Table 1. Results for analysis types.

Набор тестов	**Пройдено**	**Не пройдено**	**Доля пройденных**
Синтаксический анализ	106	2	98.1%
Анализ графа вызовов	42	0	100.0%
Анализ потоков данных	49	0	96.0%
Символьное исполнение	1313	71	94.9%
Анализ помеченных данных	95	3	96.9%
Все тесты	1650	76	95.5%

Инструмент SharpChecker успешно прошел 95.5% синтетических тестов, созданных для языка Visual Basic .NET. В то же время, для тестов на языке C# доля пройденных тестов составляет 98.5%. Таким образом, удалось добавить поддержку для большинства сценариев возникновения ошибок, рассмотренных в тестовом наборе.

4.2 Тестирование на реальных проектах

Для тестирования анализа в реальных условиях была собрана выборка проектов с открытым исходным кодом, состоящая из 5 проектов суммарным объемом 1.5 миллиона строк кода на языке Visual Basic .NET. Кроме того, в выборке присутствует и код на языке C#.

Тестирование проводилось на компьютере, оборудованном процессором Intel® Core™ i7-6700 и 32 гигабайтами оперативной памяти. Анализ набора проектов занял 2 часа, 4 минуты и 47 секунд, было сгенерировано 15816 предупреждений, из которых 7926 относятся к исходному коду на языке Visual Basic .NET, а оставшиеся 7890 к коду на C#. Часть полученных предупреждений была проанализирована и размечена вручную. Всего было размечено 1161 предупреждения, относящихся к различным детекторам. Среди них было обнаружено 977 истинных и 184 ложных предупреждений. Разметка производилась для каждого типа предупреждений по отдельности вслепую – для каждого типа случайным образом выбирался набор предупреждений для разметки. Это позволило оценить как общее качество анализа, так и качество отдельно взятых детекторов и типов анализа. Детекторы, имеющие заведомо высокую точность анализа из-за простоты их реализации (например, поиск неиспользуемых полей или забытых модификаторов `ReadOnly`), не подвергались разметке.

Таким образом, доля истинных предупреждений составила 84.2%. При этом для анализа языка C# доля истинных предупреждений составляет 88.7%. Сравнение доли истинных предупреждений для отдельных типов анализа приведено в табл. 2.

Из таблицы видно, что в среднем качество анализа для языка Visual Basic .NET ниже, чем качество для C#, однако достаточно высоко для использования инструмента в промышленных целях. Худшее качество можно объяснить тем, что детекторы были

доработаны таким образом, чтобы не изменить поведения в случае анализа кода C#, что наложило ограничения на обработку конструкций Visual Basic .NET.

Табл. 2. Сравнение качества анализа по отдельным типам анализа.
Table 2. Comparison of the quality of analysis for individual types of analysis

Тип анализа	**Visual Basic .NET Доля истинных, %**	**C# Доля истинных, %**
Синтаксический анализ	100.0%	94.6%
Анализ графа вызовов	98.4%	90.1%
Анализ потоков данных	71.5%	91.9%
Символьное исполнение	76.5%	79.5%
Анализ помеченных данных	83.2%	94.7%

Кроме того, было обнаружено более 200 межъязыковых предупреждений. В таких предупреждениях причина возникновения ошибки и место самой ошибки находятся в исходном коде на разных языках. В частности, были обнаружены предупреждения о разыменовании null, основанные на статистике вызовов метода. После добавления анализа языка Visual Basic .NET, SharpChecker начал обнаруживать вызовы библиотечных методов не только в исходном коде на C#, но и на Visual Basic .NET, что позволило построить более корректную статистику и повысить точность анализа.

Сравнение с другими статическими анализаторами не производилось из-за закрытости последних или непоказательности сравнения. В частности, ReSharper и Kiuwan предназначены для задач, отличных от задач инструмента SharpChecker.

Рассмотрим некоторые из предупреждений, обнаруженных в исходном коде на языке Visual Basic .NET.

4.2.1 UNREACHABLE_CODE

Предупреждение на листинге 1 о недостижимом коде было обнаружено в проекте `EVE-IPH` [23] в файле `ManufacturingFacility.vb` в строке 2026.

```
 1 Public Class ManufacturingFacility
 2     Private Sub LoadFacilities(...)
 3         If C1 Or C2 Or C3 Or C Then
 4             ' CODE
 5         Else
 6             If Not C Then
 7                 ' CODE
 8             Else
 9                 ' UNREACHABLE_CODE
10             End If
11         End If
12     End Sub
13 End Class
```

Листинг 1. Недостижимый код в проекте EVE-IPH.
Listing 1. Unreachable code in EVE-IPH project.

В данном примере `C1`, `C2`, `C3` и `C` – некоторые условия, не значимые в рамках рассматриваемого примера. В случае, если условие, проверяемое в строке 3, истинно, исполнение не может дойти до строки 9. В противном случае, если это условие ложно, то можно сделать вывод, что каждая из частей условия `C1`, `C2`, `C3` и `C` – ложна. Таким образом, в случае входа в ветку `Else` в строке 5 условие `Not C` будет истинно всегда. Отсюда можно сделать вывод, что код, расположенный в ветке `Else` в строке 8 будет недостижим при любых путях исполнения программы.

4.2.2 DEREF_AFTER_NULL

Предупреждение о разыменовании объекта после проверки его на равенство `Nothing`, приведенное на листинге 2, было обнаружено в компиляторе `Roslyn` версии 3.2.0, в строке 1909 файла `Binder_Lookup.vb`.

```
1 If cont IsNot Nothing And cont.SpecialType = ... Then
2     Return
3 End If
```

Листинг 2. Разыменование Nothing в проекте Roslyn.
Listing 2. Nothing dereference in Roslyn project.

В данном примере производится сравнение объекта `cont` со значением `Nothing`. Логика данного сравнения предполагает, что вторая часть выражения будет вычислена только при истинности первой части выражения. Однако оператор `And` в Visual Basic .NET не ленивый и безусловно вычисляет оба своих операнда. Таким образом, вне зависимости от результата сравнения произойдет разыменование `cont`. В данном случае следует использовать ленивый оператор `AndAlso` для предотвращения возможной ошибки.

4.3 Быстродействие

Для сравнения быстродействия использовался проект с открытым исходным кодом Roslyn – платформа для компиляции и анализа исходного кода на языках C# и Visual Basic .NET [4]. Данный проект содержит 2.2 млн. строк кода на языке C# и 1.5 млн. строк кода на языке Visual Basic .NET.

Для сравнения был произведен анализ проекта с отключенным и с включенным анализом Visual Basic .NET. Результаты тестирования приведены в табл. 3.

Табл. 3. Корреляция объема кода и времени анализа.
Table 3. Correlation between code size and analysis time.

Параметры анализа	Количество строк	Время анализа
Анализ C#	2.2 млн	0:36:04
Анализ C# и Visual Basic .NET	3.7 млн	0:46:40
Прирост	68.1%	29.4%

Прирост времени анализа в процентном соотношении значительно ниже прироста объема анализируемого кода в процентном соотношении. Разница прироста может объясняться большей по сравнению с C# «многословностью» языка Visual Basic .NET.

Быстродействие анализа исходного кода на языке Visual Basic .NET было замерено для набора из 5 проектов с открытым исходным кодом суммарным объемом 1.6 млн. строк кода. Результаты тестирования приведены в табл. 4 вместе с объемами анализируемых проектов.

Кроме того, в таблице для проекта Roslyn дополнительно приведены замеры времени для анализа исходного кода на языках C# и Visual Basic .NET по отдельности.

Из таблицы видно, что время анализа не коррелирует с ростом объема кода. Это объясняется значительными различиями в анализируемых проектах. В табл. 5 приведено распределение времени по отдельным стадиям анализа для проектов Roslyn и EVE-IPH. Распределения для остальных проектов непоказательны, так как их анализ занимает малое время.

Табл. 4. Время анализа проектов.
Table 4. Projects analysis times.

Проект	**Кол-во строк**	**Кол-во предупр.**	**Время**
DeployOffice	968	26	0:00:12
JavaRa	2.5 тыс.	110	0:00:41
QuickVB	5.8 тыс.	99	0:00:19
EVE-IPH	86.8 тыс.	2759	0:17:39
Roslyn (Visual Basic .NET)	1.5 млн.	4907	0:20:53
Roslyn (C#)	2.2 млн.	7961	0:41:44

Табл. 5. Распределение времени по стадиям анализа.
Table 5. Distribution of time by stages of analysis.

Стадия анализа	**Roslyn**		**EVE-IPH**
	C#	**Visual Basic .NET**	
Синтаксический анализ и анализ графа вызовов	16.3%	15.9%	2.2%
Символьное исполнение и анализ потоков данных	80.1%	82.9%	36.8%
Анализ помеченных данных	3.6%	1.1%	61.0%

Можно заметить, что в случае проекта Roslyn распределения времени по стадиям анализа для обоих языков сходны, что объясняется единообразностью всего проекта вне зависимости от используемого языка. Однако в случае проекта EVE-IPH большую часть времени занимает стадия анализа помеченных данных. Это связано с тем, что этот проект имеет пользовательский интерфейс и большое количество полей для пользовательского ввода. Это порождает большое количество истоков для анализа помеченных данных и значительно увеличивает время анализа в целом.

Тестирование на проектах, не содержащих исходного кода на Visual Basic .NET, показало, что время анализа для них не изменилось, несмотря на значительные изменения в большинстве детекторов.

4.4 Качество автоматической классификации предупреждений

Для упрощения анализа и разметки предупреждений в SharpChecker предоставляется механизм автоматической классификации, основанный на методах машинного обучения. Для проверки работоспособности данного механизма использовались результаты, полученные при анализе набора из 5 проектов с открытым исходным кодом общим объемом более 1.6 млн. строк кода. Для оценки качества автоматической классификации были отобраны только размеченные предупреждения. Далее эти предупреждения были разделены на две выборки – обучающую и тестовую.

В процессе выполнения статического анализа был произведен сбор метрик для классов, методов, полей, свойств, а также различных локальных объявлений. Далее полученные метрики и обучающая выборка предупреждений были переданы механизму классификации для обучения. После обучения было выполнено предсказание на тестовой выборке предупреждений. Результаты предсказаний приведены в табл. 6.

Табл. 6. Результаты классификации для Visual Basic .NET.
Table 6. Classification results for Visual Basic .NET.

Результат классификации	**Истинные**	**Ложные**
Классифицированы как истинные	167	6
Классифицированы как ложные	24	40

Таким образом, доля верно классифицированных истинных предупреждений составила 87.4%, ложных 87.0%. Для языка C# доли верно предсказанных истинных и ложных предупреждений составляют соответственно 95.4% и 79.9%. Сравнение качества предсказаний для языков Visual Basic .NET и C# приведено в таблице 7.

Табл. 7. Сравнение результатов с предсказанием для C#.
Table 7. Comparison of results with prediction for C#.

Метрика	**Visual Basic .NET**	**C#**
Точность	87.3%	92.1%
Полнота	87.4%	95.4%
F_1-мера	87.4%	93.7%

Более низкое качество предсказания по сравнению с C# можно объяснить малым количеством размеченных предупреждений. Однако результаты позволяют сделать вывод, что механизм автоматической классификации применим к статическому анализу языка Visual Basic .NET.

5. Заключение

В рамках работы инструмент SharpChecker был доработан для поддержки статического анализа языка Visual Basic .NET. Разработанные методы адаптации детекторов позволили производить поиск большинства поддерживаемых для C# типов ошибок.

В отличие от прототипа, в данной работе была добавлена полная поддержка специфики языка Visual Basic .NET на всех этапах анализа и был реализован механизм сбора метрик исходного кода. Кроме того, было проведено масштабное тестирование качества статического анализа на наборе реальных проектов с открытым исходным кодом.

При проверке работы на реальных проектах суммарным объемом более 1.6 миллиона строк удалось обнаружить 7926 предупреждений. Доля истинных предупреждений превысила 84%, что сопоставимо с результатами для C#. Добавление поддержки языка Visual Basic .NET в промышленный статический анализатор SharpChecker не ухудшило его быстродействие на проектах, не содержащих исходного кода на языке Visual Basic .NET. Кроме того, были обнаружены межъязыковые предупреждения, связанные с исходным кодом на C# и на Visual Basic .NET. Также стоит отметить, что благодаря реализации механизма сбора метрик, стала возможна автоматическая разметка предупреждений для языка Visual Basic .NET.

Список литературы / References

[1]. Википедия. Visual Basic – Википедия, свободная энциклопедия, 2025. URL: https://ru.wikipedia.org/?curid=7394&oldid=143772950. [online; accessed 14.04.2025].

[2]. TIOBE Index for ranking the popularity of Programming languages, 2025. URL: https://www.tiobe.com/tiobe-index. [online; accessed 15.05.2025].

[3]. Википедия. Visual Basic .NET – Википедия, свободная энциклопедия, 2024. URL: https://ru.wikipedia.org/?curid=17042&oldid=140564863. [online; accessed 14.04.2025].

[4]. dotnet/roslyn: The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs. URL: https://github.com/dotnet/roslyn. [online; accessed 15.05.2025].

[5]. ГОСТ Р 56939-2024. Защита информации. Разработка безопасного программного обеспечения. Общие требования. 2024. URL: https://protect.gost.ru/document1.aspx?control=31&id=263523. [online; accessed 15.05.2025].

[6]. В. С. Карцев, В. Н. Игнатьев. Поддержка Visual Basic. NET в статическом анализаторе SharpChecker. Труды ИСП РАН, том 36, вып. 3, 2024 г., стр. 49–62. DOI: 10.15514/ISPRAS-2024-36(3)-4. / Karcev V.S., Ignatiev V.N. Support of Visual Basic .NET in SharpChecker Static Analyzer. Trudy ISP RAN/Proc. ISP RAS, 2024; vol. 36, issue 3, pp. 49-62 (in Russian). DOI: 10.15514/ISPRAS-2024-36(3)-4.

[7]. V. K. Koshelev, V. N. Ignatiev, A. I. Borzilov, A. A. Belevantsev. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, 43(4):268–276, 2017.

[8]. М. В. Беляев, Н. В. Шимчик, В. Н. Игнатьев, А. А. Белеванцев. Сравнительный анализ двух подходов к статическому анализу помеченных данных. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 99–116. DOI: 10.15514/ISPRAS-2017-29(3)-7. / Belyaev M.V., Shimchik N.V., Ignatyev V.N., Belevantsev A.A. Comparative analysis of two approaches to the static taint analysis. Trudy ISP RAN/Proc. ISP RAS, 2017, vol. 29, issue 3, pp. 99-116 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-7.

[9]. А. А. Белеванцев, Е. А. Велесевич. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 53–64. DOI: 10.15514/ISPRAS-2015-27(2)-4. / Belevantsev A., Velesevich E. Analyzing C/C++ code entities and relations for program understanding. Trudy ISP RAN/Proc. ISP RAS, 2015, vol. 27, issue 2, pp. 53–64. DOI: 10.15514/ISPRAS-2015-27(2)-4.

[10]. U. V. Tsiazhkorob, V. N. Ignatyev. Classification of Static Analyzer Warnings using Machine Learning Methods. 2024 Ivannikov Memorial Workshop (IVMEM):69–74, 2024. DOI: 10.1109/IVMEM63006.2024.10659704.

[11]. В. С. Карцев, В. Н. Игнатьев. Повышение точности статического анализа за счет учета значений полей класса, имеющих единственное константное значение. Труды ИСП РАН, том 34, вып. 6, 2022 г., стр. 29–40. DOI: 10.15514/ISPRAS-2022-34(6)-2. / Karcev V.S., Ignatiev V.N. Improving the accuracy of static analysis by accounting for the values of class fields that can have only one constant value. Trudy ISP RAN/Proc. ISP RAS, 2022, vol. 34, issue 6, pp. 29-40. DOI: 10.15514/ISPRAS-2022-34(6)-2.

[12]. Wikipedia contributors. List of tools for static code analysis – Wikipedia, The Free Encyclopedia, 2024. URL: https://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&oldid=1218561224. [online; accessed 15.05.2025].

[13]. W. Wei, M. Yunxiu, H. Lilong, B. He. From source code analysis to static software testing. In 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA), 1280–1283. IEEE, 2014.

[14]. E. Firouzi, A. Sami. Visual Studio Automated Refactoring Tool Should Improve Development Time, but ReSharper Led to More Solution-Build Failures. В 2019 IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT), 2–6. IEEE, 2019.
[15]. ReSharper Features, 2025. URL: https://www.jetbrains.com/ru-ru/resharper/features/. [online; accessed 15.05.2025].
[16]. V. Lenarduzzi, F. Lomio, H. Huttunen, D. Taibi. Are SonarQube Rules Inducing Bugs? В 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 501–511. IEEE, 2020.
[17]. G. A. Campbell, P. P. Papapetrou. SonarQube in action. Manning Publications Co., 2013.
[18]. VB.NET static code analysis | SonarQube, 2025. URL: https://rules.sonarsource.com/vbnet/. [online; accessed 15.05.2025].
[19]. Common Vulnerabilities | Kiuwan, 2025. URL: https://www.kiuwan.com/common-vulnerabilities/. [online; accessed 15.05.2025].
[20]. IOperation Interface (Microsoft.CodeAnalysis) | Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.ioperation?view=roslyn-dotnet-4.13.0. [online; accessed 19.05.2025].
[21]. R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi. A Survey of Symbolic Execution Techniques. ACM Comput. Surv., 51(3), 2018. DOI: 10.1145/3182657.
[22]. 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., Ignatyev V., Borzilov A. C# static analysis framework. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 1, pp. 21-40 (in Russian). DOI: 0.15514/ISPRAS-2016-28(1)-2.
[23]. EVE Isk per Hour. URL: https://eveiph.github.io/. [online; accessed 15.04.2025].

Информация об авторах / Information about authors

Вадим Сергеевич КАРЦЕВ – аспирант Физтех-школы Радиотехники и Компьютерных Технологий МФТИ, сотрудник ИСП РАН. Научные интересы: компиляторные технологии, статический анализ программ, статическое символьное выполнение, поиск дефектов в исходном коде.

Karcev Vadim KARCEV – postgraduate student of the Phystech School of Radio Engineering and Computer Technologies of MIPT, employee of the ISP RAS. Research interests: compiler technologies, static program analysis, static symbolic execution, searching for defects in source code.

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 Nikolaevich IGNATIEV – Cand. Sci. (Phys.-Math.), Senior Researcher at the ISP RAS, Associate Professor at the Department of System Programming at the Faculty of Computational Mathematics and Cybernetics at Moscow State University. Research interests include methods for finding errors in software source code based on static analysis.