DOI: 10.15514/ISPRAS-2026-38(2)-7

Повышение полноты статического анализа приложений, использующих фреймворки, методом генерации эквивалентного кода

Бородавко Н. А., ORCID: 0009-0008-9668-6321 <nborodavko@ispras.ru>
Игнатьев В. Н., ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Московский государственный университет имени М.В. Ломоносова,
Россия, 119991, Москва, Ленинские горы, д. 1.

Аннотация. В статье описывается подход к повышению полноты анализа за счет генерации эквивалентного кода из внешних файлов. Метод восстанавливает неявные связи между элементами пользовательского интерфейса и обработчиками событий путем анализа разметки, извлечения семантических связей и генерации эквивалентного кода на C#, который моделирует сценарии взаимодействия с пользователем. Этот метод был реализован в статическом анализаторе SharpChecker с акцентом на приложения WPF. Он включает в себя синтаксический анализ XAML, извлечение ViewModel для шаблонов MVVM и генерацию кода на основе CodeDom, имитирующую обработку событий и вызовы команд в ответ на действия пользователя. Эксперименты в пяти проектах WPF с открытым исходным кодом выявили 84 новых положительных предупреждений, включая ошибки деления на ноль и ранее пропущенные разыменования нулевых указателей. Этот подход обеспечивает минимальные затраты на анализ и легко интегрируется с существующими анализаторами. Результаты демонстрируют, что генерация эквивалентного кода значительно повышает полноту статического анализа использующих фреймворки приложений.

Ключевые слова: статический анализ; язык программирования C#; повышение полноты; фреймворки; внешние файлы; обработка событий; паттерн MVVM; генерация кода.

Для цитирования: Бородавко Н.А., Игнатьев В.Н. Повышение полноты статического анализа приложений, использующих фреймворки, методом генерации эквивалентного кода. Труды ИСП РАН, том 38, вып. 2, 2026 г., стр. 95–110. DOI: 10.15514/ISPRAS–2026–38(2)–7.

Improving the recall of static analysis of applications using frameworks by generating equivalent code

N.A. Borodavko, ORCID: 0009-0008-9668-6321 <nborodavko@ispras.ru>
V.N. Ignatiev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. Static analysis of .NET applications using frameworks such as WPF and Entity Framework Core faces significant challenges due to incomplete call graphs. These frameworks rely on declarative markup files (XAML, Razor) processed at runtime, creating semantic gaps invisible to traditional static analyzers that work only with C# source code. This paper describes the approach to increase analysis recall through equivalent code generation from specialized files. The method reconstructs implicit connections between user interface elements and event handlers by parsing markup, extracting semantic relationships, and generating equivalent C# code that models user interaction scenarios. The technique was implemented in the SharpChecker static analyzer with focus on WPF applications. It includes XAML parsing, ViewModel extraction for MVVM patterns, and CodeDOM-based code generation simulating button clicks and command invocations. Experiments on five open-source WPF projects revealed 84 new true positive warnings, including division-by-zero errors and null pointer dereferences previously missed. The approach adds minimal analysis overhead and integrates seamlessly with existing analyzers. The results demonstrate that equivalent code generation significantly enhances static analysis of framework-based applications.

Keywords: static analysis; C#; improving the recall; frameworks; external files; event handling; MVVM pattern; code generation.

For citation: Borodavko N.A., Ignatiev V.N. Improving the recall of static analysis of applications using frameworks by generating equivalent code. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 2, 2026, pp. 95-110 (in Russian). DOI: 10.15514/ISPRAS-2026-38(2)-7.

1. Введение

В современной разработке программного обеспечения на платформе .NET [1] широко применяются специализированные фреймворки, такие как WPF [2] и Avalonia [3] для разработки пользовательских графических интерфейсов, ASP.NET Core [4] для веб-приложений, Entity Framework Core [5] для взаимодействия с базами данных и MAUI [6] для создания мобильных приложений. Данные фреймворки предоставляют удобные абстракции для быстрой разработки, но одновременно вызывают большие сложности в статическом анализе исходного кода.

Значительная часть логики приложений, написанных с использованием этих фреймворков, скрыта в специализированных файлах, которые игнорируются статическими анализаторами кода, потому что не содержат исходный код. Например, в WPF-приложениях разметка интерфейса и логика обработки событий описываются в файлах на языке XAML, в веб-приложениях ASP.NET Core – в Razor-файлах. Данные файлы не являются исполняемым C#-кодом и обрабатываются фреймворками динамически во время выполнения приложения. Как итог, традиционные статические анализаторы, работающие исключительно с исходным C#-кодом, не способны восстановить полную последовательность вызовов методов, инициируемых через обработчики событий или сгенерированный фреймворком код.

Неполнота графа вызовов приводит к двум категориям ошибок. Во-первых, могут возникать ложные предупреждения, когда анализатор неверно классифицирует методы как «неиспользуемые», поскольку не способен обнаружить их вызовы. Во-вторых, ошибки могут быть пропущены, поскольку анализатор не может смоделировать возможные пути выполнения программы.

В частности, для приложений на Spring Framework имеются специализированные статические анализаторы, такие как Jasmine, которые демонстрируют сложности анализа, возникающие из-за внедрения зависимостей и рефлексии, широко используемых фреймворками в современных корпоративных приложениях [7]. Аналогичные проблемы приводят к потере полноты анализа при применении статических методов к архитектуре современных Java сервисов и Android-приложений [8].

В качестве ошибочной ситуации, не обнаруживаемой статическими анализаторами, можно привести следующий демонстрационный пример. Пусть в главном окне WPF-приложения содержатся слайдер и кнопка, при изменении положения слайдера вызовется метод UpdateValue, где обновится значение переменной value, а при нажатии на кнопку будет вызван метод ButtonClick, в котором вычисляется результат деления 10 / value, после чего сама переменная value обнуляется. В листинге 1 приведен класс, реализующий алгоритм работы приложения.

Данный пример демонстрирует потенциальную ошибку: сдвиг слайдера на ноль и нажатие кнопки или двойное нажатие кнопки подряд приведет к делению на ноль. Однако разметка интерфейса с указанием обработчиков событий для элементов находится в отдельном XAML-файле (листинг 2), и статический анализатор, работающий исключительно с C#, не распознает связи между нажатием кнопки и вызовом обработчика.

```
 1  public partial class MainWindow : Window
 2  {
 3      private int value = 1;
 4
 5      public void UpdateValue(object sender,
 6          RoutedPropertyChangedEventArgs<double> e)
 7      {
 8          value = Convert.ToInt32(e.NewValue);
 9      }
10
11      public void ButtonClick(object sender, RoutedEventArgs e)
12      {
13          int res = 10 / value;
14          // возможная ошибка деления на ноль
15
16          value = 0;
17      }
18  }
```

Листинг 1. Пример пропущенной статическим анализатором ошибки «деление на ноль».
Listing 1. Example of an «division by zero» error missed by the static analyzer.

```
1  <Window
2    x:Class="MyApp.MainWindow" Title="MainWindow"
3    Height="350" Width="525">
4    <Grid>
5      <Slider Value="0" ... ValueChanged="UpdateValue"/>
6      <Button Content="Reset" Click="ButtonClick"/>
7    </Grid>
8  </Window>
```

Листинг 2. XAML-код разметки демонстрационного приложения.
Listing 2. XAML markup code for the demo application.

Без явного представления этих связей в графе вызовов анализатор не может выявить сценарий, в котором кнопка нажата дважды подряд, что вызывает исключение деления на ноль. Таким образом, критическая ошибка останется незамеченной до этапа выполнения.

Данное WPF-приложение было написано с использованием событийного подхода. Однако в последнее время разработчики все чаще используют архитектурный паттерн MVVM (Model-View-ViewModel) [9], при котором логика отделяется от интерфейса, что добавляет дополнительные сложности в анализе программ.

Перечисленные проблемы приводят к большому количеству ложных предупреждений и пропущенным ошибкам, поэтому актуальна разработка методов их поддержки в статическом анализаторе. Подобно другим работам по статическому анализу, использующим промежуточное представление программы [10], в данной работе для повышения полноты анализа применяется преобразование исходных артефактов в форму, удобную для дальнейшей обработки. Метод генерации эквивалентного кода представляет собой универсальный и воспроизводимый подход, который может быть адаптирован к любому фреймворку, использующему специализированные файлы разметки.

В данной работе метод генерации эквивалентного кода будет рассмотрен в рамках реализации поддержки приложений WPF для инструмента SharpChecker, предназначенного для поиска ошибок в объемных проектах на платформе .NET методами статического анализа. Приведенный алгоритм может быть применим и для других фреймворков с незначительными изменениями.

1.1 Цели и задачи

Цель работы – повысить полноту статического анализа приложений, использующих фреймворки, путем применения метода генерации эквивалентного кода из специализированных внешних файлов.

Результатом работы должен стать универсальный метод, который будет:

- увеличивать полноту графа вызовов путем явного представления в исходном коде связей между элементами интерфейса (XAML) и обработчиками событий (C#);
- моделировать пользовательские сценарии работы с приложением, позволяя анализатору выявить ошибки, зависящие от порядка выполнения;
- обеспечивать воспроизводимость анализа – в отличие от динамического анализа, результаты полностью детерминированы и не зависят от конкретных тестовых сценариев;
- интегрироваться с существующими анализаторами без изменения их архитектуры – сгенерированный код добавляется в сборку как обычный C#-код.

1.2 Обзор существующих решений

Некоторые анализаторы включают поддержку анализа внешних файлов, однако эта поддержка остаётся весьма ограниченной. В качестве примера можно привести инструмент Coverity [11], корпоративный статический анализатор, который для веб-приложений на платформе ASP.NET предоставляет поддержку анализа файлов разметки (.aspx) и Razor-шаблонов (.cshtml), включая обнаружение уязвимостей безопасности, таких как XSS и SQL-инъекции. Однако Coverity фокусируется преимущественно на анализе безопасности веб-форм, в то время как восстановление полного графа вызовов в проекте остается ограниченным.

Следующие статические анализаторы также рассматривают специальные файлы, но исключительно на уровне синтаксической валидации. Например, в SonarQube [12] после запросов сообщества разработчиков была введена поддержка анализа внешних файлов.

Однако эта поддержка ограничивается анализом XAML как XML-документов через SonarXml, то есть проверяется корректность XML-структуры, баланс тегов, валидность пространств имен, но ничего более. Проверки SonarQube для XAML не анализируют соответствие обработчиков событий между разметкой и кодом и не дополняют граф вызовов.

Другим примером выступает плагин для Visual Studio ReSharper [13]. Он предоставляет встроенную поддержку XAML-файлов с проверками, которые в основном сосредоточены на синтаксической корректности. ReSharper может провести только базовую проверку: существует ли метод с именем обработчика события в связанном классе. Помимо этого, при работе с паттерном MVVM и привязками данных возникает проблема: ReSharper часто не может определить тип контекста данных на момент анализа, что приводит к ложным предупреждениям, даже когда привязка технически корректна и будет работать во время выполнения. Простая синтаксическая поддержка недостаточна для повышения полноты графа вызовов.

Некоторые инструменты содержат специализированные диагностики для проблем, типичных для фреймворков, но при этом все еще игнорируют внешние файлы. PVS-Studio [14] предоставляет специализированные для WPF диагностики, которые помогают разработчикам избежать опечаток и логических ошибок в регистрации зависимых свойств. Однако эта поддержка остается локальной: анализируется только сам код определения свойства, а не связь между свойством и его использованием в XAML.

В научных работах, связанных с аналогичными по архитектуре фреймворками на других платформах (Android, Spring) [8, 15–18], исследователи предлагают в качестве решения проблемы использование динамического анализа. Этот метод предполагает выполнение приложения под мониторингом и отслеживание фактических вызовов методов во время работы. Теоретически, динамический анализ может обнаружить все реально выполняемые пути кода, включая те, что инициируются из XAML.

Однако динамический анализ страдает от критических ограничений. Во-первых, результаты анализа полностью зависят от конкретных сценариев тестирования и входных данных. Если тестовый набор не включает определенную последовательность пользовательских действий, ошибка в этом пути останется необнаруженной. Во-вторых, накладные расходы на инструментирование кода и мониторинг выполнения значительны, что делает анализ медленным и затратным по ресурсам. В-третьих, результаты динамического анализа часто страдают от проблем с воспроизводимостью – одна и та же программа может демонстрировать различное поведение при повторных запусках из-за недетерминированности во времени, условий многопоточности или случайных компонентов.

1.3 Общие механизмы современных фреймворков пользовательских интерфейсов

Современные фреймворки для разработки пользовательских интерфейсов (графические интерфейсы, веб-интерфейсы, мобильные приложения) основаны на нескольких общих архитектурных принципах и механизмах, которые повторяются независимо от конкретной платформы.

В событийно-управляемой архитектуре MVC (Model-View-Controller) приложение разделяется на три компонента: модель, представление и контроллер.

- Модель содержит данные приложения и правила их обработки. Она реагирует на команды из контроллера, изменяет свое состояние, и передает данные в представление.
- Представление визуализирует в интерфейсе полученную от модели информацию. Для представления как правило используются специализированные языки (XML-подобные языки, JSX, HTML), позволяющие декларативно описывать структуру интерфейса, что позволяет отделить разметку от логики обработки взаимодействий пользователя с приложением.
- Контроллер определяет, как приложение будет реагировать на действия пользователя. Взаимодействие с элементами интерфейса (нажатие кнопки, ввод текста, выбор элемента и т.д.) генерирует события, для которых разработчик регистрирует функции-обработчики. Регистрация обработчиков обычно происходит в представлении для каждого элемента интерфейса, а сами действия описываются в контроллере. События могут быть направлены, распространяться вверх по иерархии элементов или обрабатываться локально на конкретном элементе.

Ввиду высокой сложности поддержки приложений с использованием событийного подхода из-за тесной связи бизнес-логики и UI-логикой разработчики в настоящее время чаще всего используют архитектуру MVVM (Model-View-ViewModel), где вместо контроллера используется модель представления. Данная абстракция является посредником между моделью и представлением, где содержатся данные из модели, преобразованные для отображения в представлении, и команды для взаимодействия, которые используются представлением для влияния на модель. Такой подход обеспечивает лучшую чистоту кода и позволяет дизайнерам и разработчикам работать независимо.

Класс модели представления явно устанавливается в представлении (либо в соответствующем классе представления) в качестве контекста данных, который определяет объект, свойства которого доступны для привязки в данной части интерфейса.

Подготовленные для отображения данные из модели связываются с соответствующими элементами управления из представления через механизм привязки данных, в рамках которого происходит синхронизация между состоянием модели представления и представлением этих данных в интерфейсе. Изменения в модели автоматически отражаются в интерфейсе, и наоборот. Привязка данных часто использует выражения привязки (binding expressions), которые связывают свойства элементов интерфейса с соответствующими свойствами модели представления.

Вместо прямого связывания событий с методами обработчиков используется паттерн команд. Команда – это объект, инкапсулирующий действие и его параметры. Элементы интерфейса связываются с командами через привязку данных, обеспечивая разделение между логикой обработки и представлением.

Все эти механизмы направлены на обеспечение разделения ответственности и чистоты кода, но одновременно они создают проблему для статического анализа: связи между интерфейсом (определенным в специализированных файлах) и логикой (определенной в коде) становятся опосредованными, неявными и динамическими.

1.4 Архитектура фреймворка WPF и подходы разработки приложений

WPF (Windows Presentation Framework) – один из главных фреймворков платформы .NET для разработки клиентских Windows-приложений с графическим интерфейсом [2, 19]. В качестве специализированного языка разметки WPF использует XAML (eXtensible Application Markup Language) [20] – язык на основе XML, позволяющий декларативно создавать объекты интерфейса и организовывать их в иерархии.

В WPF используются два подхода к созданию приложений – событийный (MVC) и паттерн MVVM.

Событийный подход основан на прямой обработке событий элементов управления (например, нажатие кнопки) в code-behind (файл `.xaml.cs`), где логика тесно связана с интерфейсом.

В подходе через паттерн MVVM обработчики логики отделены от интерфейса. Действия пользователя связываются с командами (объектами типа `ICommand`, например

`RelayCommand`), которые определены в классе модели представления (ViewModel). Привязка данных в XAML указывает на команды через выражения привязки, например `Command="{Binding SaveCommand}"`. Контекст привязки (`DataContext`) указывает на экземпляр класса модели представления, который используется для разрешения привязок во время выполнения приложения. Он может задаваться либо напрямую в коде XAML, либо в конструкторе класса окна.

Оба подхода создают одинаковую проблему для статического анализа: информация о связях между интерфейсом и кодом остается недоступной для анализаторов, работающих только с исходным кодом на C#.

2. Предложенный метод

2.1 Схема работы алгоритма

Предложенный метод состоит из четырех шагов:

1. поиск и разбор внешних файлов;
2. анализ специализированных файлов;
3. генерация эквивалентного C#-кода с моделированием последовательностей пользовательских действий;
4. анализ сгенерированного кода по аналогии с существующим.

На вход методу подается анализируемый проект, на выходе возвращается проект со сгенерированными файлами.

2.1.1 Поиск и разбор внешних файлов

На первом этапе инструмент проводит поиск специализированных файлов в проекте во время процесса подготовки к анализу. Информацию о том, в каких каталогах их следует искать, можно взять из файлов проекта (`.csproj`) или при перехвате сборки, когда анализатор напрямую интегрирован с системой сборки на этапе, предшествующем трансформации специализированных файлов компилятором в промежуточные форматы (например, BAML для XAML), что позволяет получить доступ к исходным файлам до их обработки и преобразования. Инструмент просматривает список файлов проекта, предоставляемый системой сборки, и идентифицирует файлы с соответствующими расширениями (например, `.xaml` для WPF).

Для каждого найденного файла выполняется разбор с целью извлечения семантически значимой информации. Процесс строит иерархическое дерево элементов, сохраняя информацию о типе каждого элемента, его свойствах и дочерних элементах.

Результат обработки каждого внешнего файла – структурированное представление специализированного файла, пригодное для дальнейшего анализа.

2.1.2 Анализ специализированных файлов

На втором этапе выполняется анализ полученной информации для определения связей между интерфейсом и кодом. Этап включает два подэтапа:

- Сопоставление обработчиков событий и команд к элементам интерфейса: для каждого элемента в специализированном файле анализируются определенные события и команды. Из описания интерфейса извлекаются имена обработчиков и команд. Эта информация используется для идентификации методов в коде, которые будут выполнены при взаимодействии пользователя с интерфейсом;
- Определение контекста данных: если приложение использует архитектурный паттерн MVVM, на этом подэтапе определяется класс, который служит контекстом привязки. Для этого анализируется соответствующий данному специальному файлу класс для поиска присваиваний контексту. Также разбирается найденный ViewModel-класс для определения команд, доступных для привязки.

2.1.3 Генерация эквивалентного C#-кода

Поскольку внешние файлы сами по себе не являются исполняемым кодом, для статического анализа их содержимое трансформируется в эквивалентный класс. Идея состоит в том, чтобы сгенерировать исходный код, описывающий те же элементы интерфейса, что и в разметке. На практике это означает создание C#-классов, где объявляются поля для элементов управления (кнопки, поля текстового ввода) и регистрируются связанные с ними события. Кроме того, если используется архитектурный паттерн MVVM, то генерируется код, который создает экземпляры класса логики и регистрирует команды.

При генерации эквивалентного кода метод преобразует структуру специализированного файла (такого как XAML) в эквивалентный исходный код на языке C#, который воспроизводит объектную иерархию и связи между элементами интерфейса. Первый этап заключается в разборе специализированного файла и извлечении всей структурной информации: типов элементов управления, их свойств, привязок данных, событий и команд. На основе полученной информации генератор кода создает статический метод для построения интерфейса (например, `WindowCreate()`), который программно воссоздает всю визуальную структуру, инстанцируя элементы управления и устанавливая их свойства, вложенность и привязки. Это позволяет статическому анализатору работать с полным графом всех элементов интерфейса в виде обычного C#-кода.

Однако главный компонент метода – это моделирование пользовательских сценариев. Каждый сценарий представляет собой отдельную функцию, которая воспроизводит возможные последовательности пользовательских действий. В рамках каждого сценария генерируется последовательность вызовов: сначала создается новый экземпляр окна через вызов статического метода для построения интерфейса (что обеспечивает чистое, изолированное состояние), затем инициализируется контекст данных для данного окна (если используется паттерн MVVM), и после этого последовательно вызываются обработчики событий или методы команд в различных комбинациях. Критически важно, что окно и контекст данных создаются заново для каждого сценария – это гарантирует, что состояние, измененное одним сценарием, не влияет на другие. Такая изоляция позволяет анализатору независимо исследовать каждый потенциальный путь выполнения без смешиваний состояний между различными последовательностями действий.

Критическим аспектом является проблема комбинаторного взрыва. Пусть N – количество всех пар «элемент-обработчик/команда» в окне, а L – максимальная длина генерируемых последовательностей. Тогда количество возможных комбинаций составляет N^L, что растет экспоненциально. Для обычного окна с 10 элементами управления даже при глубине $L = 3$ число комбинаций достигает $10^3 = 1000$, что при большей длине приводит к комбинаторному взрыву. Полный перебор всех возможных цепочек событий становится нерезультативным с экспоненциальным временем работы $N^L * T_{avg}$, где T_{avg} – среднее время анализа одной комбинации, или вовсе не завершается.

Для оценки практических масштабов задачи была проведена экспериментальная оценка на тестовых проектах. В проектах с открытым исходным кодом размер интерфейсов варьируется от 1 до 50 интерактивных элементов с обработчиками на окно. При ограничении глубины $L = 2$ для окна с 20 обработчиками генерируется $20^2 = 400$ комбинаций, что обрабатывается за приемлемое время. Однако увеличение до $L = 3$ даёт уже $20^3 = 8000$ комбинаций, а при $L = 4$ – 160 тысяч комбинаций только для одного окна. В реальных проектах с десятками окон такой подход становится непрактичным.

Поэтому в данной работе анализатор ограничивает длину комбинаций значением $L = 3$, что позволяет выявить значительную часть ошибок, возникающих при простых последовательностях взаимодействий, жертвуя обнаружением более сложных сценариев с четырьмя и более действиями.

В результате получается эквивалентный C#-класс, который включается в сборку для анализа, что помогает дополнить граф вызовов множеством ребер между всеми обработчиками событий и команд.

2.1.4 Анализ проекта со сгенерированным кодом

После завершения генерации эквивалентного кода сгенерированные файлы добавляются в сборку проекта и включаются в процесс компиляции наряду с остальным исходным кодом. С точки зрения компилятора и статического анализатора, сгенерированный код ничем не отличается от обычного исходного C#-кода, что обеспечивает полную интеграцию без необходимости модификации архитектуры анализатора.

При запуске инструмента анализа на проект со встроенным сгенерированным кодом предупреждения, найденные анализатором в этом коде, будут отображаться в стандартных отчётах анализатора. Важным аспектом является то, что сгенерированный код содержит трассируемую информацию о происхождении: каждое предупреждение из методов-сценариев может быть связано с исходным XAML-файлом и соответствующей последовательностью пользовательских действий, что облегчает разработчику понимание контекста обнаруженной ошибки. Таким образом, если анализатор найдёт потенциальную ошибку, разработчик может понять, в каком конкретном сценарии взаимодействия пользователя с интерфейсом (например, при какой последовательности нажатий кнопок) может возникнуть проблема, что значительно облегчает отладку и воспроизведение ошибки.

2.2 Особенности реализации

Данный подход, как говорилось ранее, был реализован в рамках инструмента SharpChecker для поддержки статического анализа приложений, использующих фреймворк WPF. Данный статический анализатор был разработан для поиска ошибок в объемных проектах на платформе .NET [21]. Инструмент реализует различные виды анализа кода, включая анализ потока данных, проверку на использование небезопасных операций и выявление типичных ошибок программирования [22]. В частности, инструмент обладает встроенным перехватом процесса сборки через интеграцию с системой MSBuild [23], что позволяет получить доступ к промежуточным артефактам компиляции и специализированным файлам проекта.

Для реализации метода генерации эквивалентного кода в анализаторе был разработан специализированный модуль поддержки фреймворков. Модуль интегрирован в процесс сборки проекта и работает перед этапом основного анализа. Модуль состоит из нескольких компонентов:

- **Разбор XAML**: использует встроенный в .NET парсер `XamlXmlReader` для построения объектного дерева. Извлекает информацию о типах элементов, свойствах, событиях и привязках данных. Результат разбора сохраняется в иерархическом дереве для дальнейшей обработки.
- **Извлечение контекста**: определяет ViewModel-класс при использовании в приложении паттерна MVVM. Анализирует файл кода для поиска присваиваний контексту привязки и извлечения полного имени типа класса логики. Также анализирует найденный класс для определения доступных команд.
- **Генератор кода**: на основе информации, полученной предыдущими компонентами, генерирует partial-класс, содержащий инициализацию элементов, регистрацию обработчиков и методы-эмуляторы. Для генерации используется CodeDOM (Code Document Object Model), что позволяет создать синтаксически корректный исходный код на языке C#.

Важно отметить, что в контексте WPF генерация кода приобретает специфические черты. Генератор создает поля для каждого именованного элемента управления из XAML-разметки (кнопок, списков, меню и т.д.) и полностью воспроизводит древовидную структуру контейнеров (`Grid`, `StackPanel`, и др.). Для каждого элемента с привязанным событием или командой устанавливается связь: например, для кнопок с командами через `Command="{Binding CreateNode}"` генератор регистрирует обработчик события (например, `button.Click`). Что особенно важно, когда приложение использует паттерн MVVM с использованием RelayCommand, генератор создает специальный класс-обертку над оригинальным ViewModel (например, `MainWindowViewModel_SC`), который содержит методы-делегаты, ссылающиеся на оригинальные команды. Генерация отдельного ViewModel-класса вместо использования оригинального требуется потому, что, во-первых, оригинальный класс может содержать сложную логику инициализации, зависимости или состояние, которое затруднило бы анализ. Во-вторых, методы-делегаты позволяют явно связать вызовы обработчиков в сгенерированном коде, сделав связи видимыми для анализатора.

Сценарии в WPF представляют собой статические методы с генерированными именами, каждый из которых является независимым контекстом выполнения. В начале каждого метода вызывается `WindowCreate()` для создания свежего экземпляра окна, затем инстанцируется новый экземпляр сгенерированного ViewModel-класса и присваивается окну как `DataContext`. После этого последовательно вызываются методы-делегаты в различных комбинациях, моделируя реальные сценарии: например, пользователь может нажать кнопку `Create`, затем `Edit`, и затем выполнить операцию Drag-and-Drop. Каждая такая комбинация создает новый путь выполнения, который анализатор может исследовать.

Таким образом, процесс работы модуля на каждом специализированном файле в проекте выглядит следующим образом:

1. Система сборки перехватывает специализированный файл перед компиляцией. Модуль считывает исходный текст файла;
2. Данный файл разбирается и строится объектное представление, сохраняя информацию о структуре интерфейса;
3. Проводится анализ построенного представления файла, в котором собирается полная информация о связях между интерфейсом и логикой. В случае обнаружения использования паттерна MVVM (например, при обнаружении у одного из элементов характерного свойства `Command`) определяется контекст данных при помощи извлекателя контекста;
4. Создается partial-класс с инициализацией элементов, регистрацией обработчиков и методами-эмуляторами;
5. Сгенерированный код сохраняется в файл и добавляется в проект для компиляции вместе с остальным кодом.

При запуске анализатора SharpChecker сгенерированный код анализируется как обычная часть проекта, позволяя выявить ошибки в смоделированных сценариях взаимодействия.

В листинге 3 приведен пример сгенерированного кода для демонстрационного приложения (листинги 1 и 2). В `WindowCreate` инициализируются элементы интерфейса согласно разметке приложения, а `__Gen_Comb_N__` – все возможные комбинации обработчиков событий длины L (в данном примере выбран $L = 2$). Каждая комбинация находится в отдельном контексте при помощи создания объекта окна через метод `WindowCreate`.

Таким образом, в комбинации 3 получается последовательность, приводящая к делению на ноль.

```
 1  public partial class MainWindow
 2  {
 3      public Slider slider = new Slider();
 4      public Button button = new Button();
 5      public Grid grid = new Grid();
 6      public static MainWindow WindowCreate()
 7      {
 8          MainWindow window = new MainWindow();
 9          window.Content = window.grid;
10          window.grid.Children.Add(window.slider);
11          window.grid.Children.Add(window.button);
12          window.slider.ValueChanged += window.UpdateValue;
13          window.button.Click += window.ButtonClick;
14          return window;
15      }
16      // ...
17      // Приводящая к ошибке последовательность обработчиков
18      public static void @__Gen_Comb_3__()
19      {
20          MainWindow mainWindow = MainWindow.WindowCreate();
21          mainWindow.ButtonClick(mainWindow.button,
22                                 new RoutedEventArgs());
23          mainWindow.ButtonClick(mainWindow.button,
24                                 new RoutedEventArgs());
25      }
26      // ...
27  }
```

Листинг 3. Пример сгенерированного класса для демонстрационного приложения
Listing 3. Example of a generated class for a demo application

3. Результаты

В качестве тестовых приложений для измерения реальной эффективности анализа с модулем поддержки использовались проекты с открытым исходным кодом, написанные с использованием фреймворка WPF [24–28]. Общий размер кодовой базы проектов на C# – 3 млн. строк, на XAML – 76 тыс. строк.

В ходе экспериментов был выполнен сравнительный прогон статического анализатора SharpChecker на данном наборе реальных WPF-проектов в двух режимах: без модуля поддержки популярных фреймворков и с включенным модулем. Результаты сравнения числа истинных (True Positive, TP) и ложных предупреждений (False Positive, FP) до и после использования модуля поддержки отображены в табл. 1.

Время анализа данных проектов без модуля поддержки составило 5 минут 44 секунды, с включенным модулем – 6 минут 37 секунд (+15%).

В расширенном режиме было обнаружено 84 новых истинных предупреждений. Рост числа истинных предупреждений свидетельствует о значительном улучшении полноты покрытия анализатором потока управления проекта. Новые предупреждения относятся прежде всего к ошибкам деления на ноль (детекторы DIVISION_BY_ZERO) и разыменованию пустой ссылки (DEREF_OF_NULL, DEREF_AFTER_NULL), а также к уязвимостям командной

инъекции (COMMAND_INJECTION), которые без модуля поддержки оставались невидимыми анализатору. Вместе с этим 9 ложных предупреждений после использования модуля были устранены.

Табл. 1. Результаты тестирования метода на наборе открытых проектов
Table 1. The proposed method testing results on the set of open-source projects

Проект	Было предупреждений	Ушло TP	Ушло FP	Пришло TP	Пришло FP	Стало предупреждений
Chordious	699	3	4	7	9	708
ScreenToGif	1267			7		1274
JPPhotoManager	211	1	4	70		276
NVM#	60					60
PwM	127		1		389	515

При этом 4 истинных предупреждения не были обнаружены после добавления модуля поддержки. Это объясняется тем, что при генерации комбинаций последовательных вызовов обработчиков событий и команд появились новые циклы в графе вызовов. При анализе инструмент SharpChecker разбивает эти циклы и анализирует функции в отличающемся от обычного анализа порядке, что приводит к получению других резюме функций.

Одновременно с этим в расширенном режиме возросло число ложных предупреждений для отдельных правил: для DEREF_OF_NULL.ARGUMENT зарегистрировано 386 ложных предупреждений, для UNREACHABLE_CODE.EXCEPTION – 3 FP, для DEREF_AFTER_NULL – 1 FP и для DEREF_OF_NULL.RET.USER.PROC – 8 FP. Таким образом, в расширенном режиме было обнаружено 398 ложных предупреждений.

Важно отметить, что ложные предупреждения составляют 83% от общего числа новых предупреждений и сосредоточены преимущественно в одном проекте (PwM), где они составляют 75% от всех предупреждений проекта (389 из 515).

Несмотря на рост числа ложных предупреждений, практическая польза модуля поддержки значительна. Было обнаружено 84 новых истинных предупреждения об опасных ошибках (деление на ноль, разыменование нулевой ссылки, командная инъекция), которые ранее оставались незамеченными. Эти ошибки могут привести к критическим сбоям и уязвимостям безопасности в реальных приложениях. Кроме того, в четырех проектах из пяти рост ложных предупреждений был минимальным (от 0 до 9 FP), что свидетельствует о приемлемой точности метода для большинства типичных WPF-проектов.

Причина, по которой сильно увеличилось число ложных предупреждений DEREF_OF_NULL.ARGUMENT, заключается в следующем. При генерации комбинаций последовательностей событий и команд алгоритм создает множество сценариев вызовов обработчиков. В одном из проектов (PwM) это привело к генерации большого числа комбинаций, в которых моделируются невозможные или недостижимые пути выполнения. Например, определенные последовательности нажатий кнопок могут быть невозможны в реальной работе приложения, если кнопка становится недоступной (свойство `IsEnabled = false`) после определенного действия или находится в другой вкладке (мы не можем нажать последовательно на кнопки, находящиеся в разных вкладках). Анализатор, однако, при синтаксическом анализе сгенерированного кода не учитывает эти ограничения видимости и активности элементов, что приводит к появлению предупреждений детекторов на пути, которые не могут быть достигнуты в реальности. Эти 386 ложных предупреждений

сосредоточены в одном проекте и возникают из-за того, что при сгенерированных комбинациях возникает невозможное разыменование нулевого указателя.

В дальнейшем планируется улучшить алгоритм генерации комбинаций путем добавления многоуровневой фильтрации недостижимых сценариев:

- **Анализ видимости и активности элементов управления**: отслеживание свойства `Visibility` и `IsEnabled` элементов для исключения взаимодействия со скрытыми элементами;
- **Фильтрация по контексту вкладок**: анализ иерархии элементов `TabControl` для исключения последовательностей событий, затрагивающих элементы из разных вкладок;
- **Моделирование логики переключения состояний**: отслеживание изменений состояний элементов в ходе выполнения последовательностей событий для построения более точной модели достижимых путей;
- **Ограничение глубины комбинаций на основе эвристик**: адаптивное определение максимальной длины последовательностей на основе сложности проекта для предотвращения комбинаторного взрыва.

3.1 Примеры предупреждений

3.1.1 DIVISION_BY_ZERO

Рассмотрим предупреждение, обнаруженное в проекте ScreenToGif в файле `VideoSource.xaml.cs` в строке 214 (листинг 4).

```
1  private async void OkButton_Click(object sender,
2                                    RoutedEventArgs e)
3  {
4      // ...
5
6      Delay = 1000 / FpsIntegerUpDown.Value; // DIVISION_BY_ZERO
7
8      // ...
9  }
```

Листинг 4. Демонстрация предупреждения DIVISION_BY_ZERO.INPUT в проекте ScreenToGif
Listing 4. Demonstration of the DIVISION_BY_ZERO.INPUT warning in the ScreenToGif project

В данном обработчике события `OkButton_Click` при вычислении значения `Delay` в качестве делителя используется значение `FpsIntegerUpDown.Value`, которое задается пользователем в соответствующем элементе пользовательского интерфейса. Таким образом, если пользователь введет ноль и после этого нажмет на кнопку `OkButton`, будет вызван обработчик события `OkButton_Click`, где и произойдет деление на ноль.

Это предупреждение не обнаруживалось раньше, потому что анализатор не знал, что значение `FpsIntegerUpDown.Value` может измениться извне через взаимодействие пользователя с интерфейсом, и поэтому не учитывал эту возможность при построении потока управления. Метод генерации эквивалентного кода явно связывает элементы интерфейса с обработчиком события `OkButton_Click`, что позволяет анализатору понять, что данный метод вообще вызывается.

3.1.2 DEREF_OF_NULL

Следующее предупреждение было обнаружено в проекте Chordious в строке 259 файла `DiagramMarkEditorViewModel.cs` (листинг 5).

```
1  public DiagramMarkEditorViewModel(DiagramMark diagramMark,
2                                    bool isNew)
3  {
4      // ...
5
6      Style = new ObservableDiagramStyle(clone);
7      Style.MarkStyle.MarkType = DiagramMark.Type; // DEREF_OF_NULL
8
9      // ...
10 }
```

Листинг 5. Демонстрация предупреждения DEREF_OF_NULL в проекте Chordious – конструктор ViewModel-класса

Listing 5. Demonstration of the DEREF_OF_NULL warning in the Chordious project – constructor of the ViewModel class

При инициализации ViewModel-класса создается объект `ObservableDiagramStyle`, используя конструктор с одним параметром. В листинге 6 приводится сигнатура и содержимое данного конструктора.

```
1  internal DiagramMarkStyleWrapper MarkStyle { get; private set; }
2
3  public ObservableDiagramStyle(DiagramStyle diagramStyle,
4         DiagramMarkStyleWrapper diagramMarkStyle = null) : base()
5  {
6      // ...
7      if (diagramMarkStyle is not null)
8      {
9          // ...
10         MarkStyle = diagramMarkStyle;
11     }
12     // ...
13     MarkStyle.ForEachMarkType(() => { ... }); // DEREF_OF_NULL
14     // ...
15 }
```

Листинг 6. Демонстрация предупреждения DEREF_OF_NULL в проекте Chordious – конструктор ObservableDiagramStyle

Listing 6. Demonstration of the DEREF_OF_NULL warning in the Chordious project – ObservableDiagramStyle constructor

Так как второй параметр не указан, то аргумент `diagramMarkStyle` будет инициализирован `null`, что вызывает следующие проблемы. Во-первых, до момента первого разыменования поле `MarkStyle` не будет инициализировано (мы не войдем в тело `if` из-за аргумента), и в результате попытки вызова метода `ForEachMarkType` произойдет разыменование нулевого указателя. Во-вторых, второе разыменование нулевого указателя происходит уже после создания объекта `ObservableDiagramStyle`, внутри конструктора ViewModel-класса, где также происходит обращение к полю `MarkStyle`.

Эта ошибка не могла быть обнаружена раньше, так как анализатор не видит, откуда вызывается этот конструктор. В XAML-файле находятся какие-то элементы, привязаны какие-то данные, но для инструмента это абсолютно невидимо. Без модуля поддержки анализатор имеет очень ограниченную видимость всех возможных путей вызова, потому что он может увидеть только прямые вызовы конструктора в исходном C#-коде. С модулем поддержки анализатор генерирует код, который явно моделирует различные сценарии инициализации окна с разными состояниями ViewModel.

4. Заключение

В работе предложен универсальный метод повышения полноты анализа приложений, использующих фреймворки с декларативной разметкой, путем генерации эквивалентного кода. Метод успешно применен для поддержки WPF в статическом анализаторе SharpChecker и продемонстрирован на реальных проектах.

Предложенный подход демонстрирует, что генерация эквивалентного кода является эффективным и практически применимым методом для восстановления полноты анализа приложений с декларативной разметкой. Метод может быть применен к другим фреймворкам с аналогичной архитектурой без фундаментальных изменений в его структуре.

Список литературы / References

[1]. .NET Developer Platform. Available at: https://learn.microsoft.com/en-us/dotnet/, accessed 22-05-2025.

[2]. Windows Presentation Foundation for .NET documentation | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/, accessed 22-05-2025.

[3]. Avalonia UI: Cross-Platform UI Frameworks for .NET Developers. Available at: https://avaloniaui.net/, accessed: 22-05-2025.

[4]. ASP.NET Core documentation. Available at: https://learn.microsoft.com/en-us/aspnet/core/, accessed: 22-05-2025.

[5]. Entity Framework documentation hub. Available at: https://learn.microsoft.com/en-us/ef, accessed: 22-05-2025.

[6]. .NET Multi-Platform App UI (.NET MAUI). Available at: https://dotnet.microsoft.com/en-us/apps/maui, accessed: 22-05-2025.

[7]. M. Chen, T. Tu, H. Zhang, Q. Wen и W. Wang. Jasmine: A static analysis framework for spring core technologies. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-13, 2022.

[8]. J. Samhi, M. Miltenberger, M. Alecci, S. Arzt, T. Bissyand´e и J. Klein. Do you have 5 min? Improving Call Graph Analysis with Runtime Information. In Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, pp. 540-544, 2025.

[9]. S. Chakraborty и P. Aithal. MVVM Demonstration Using C# WPF. International Journal of Applied Engineering and Management Letters (IJAEML), 7(1): 1-14, 2023.

[10]. Avetisyan, S. Kurmangaleev, S. Sargsyan, M. Arutunian и A. Belevantsev. LLVM-based code clone detection framework. In 2015 Computer Science and Information Technologies (CSIT), 2015, pp. 100-104. DOI: 10.1109/CSITechnol.2015.7358259

[11]. Coverity Scan - Static Analysis. Available at: https://scan.coverity.com/, accessed: 06-11-2025.

[12]. Code Quality, Security & Static Analysis Tool with SonarQube | Sonar. Available at: https://www.sonarsource.com/products/sonarqube/, accessed: 22-05-2025.

[13]. XAML Editing Tools - Features | ReSharper. Available at: https://www.jetbrains.com.cn/en-us/resharper/features/xaml_editor.html, accessed: 22-05-2025.

[14]. Проверяем исходный код WPF примеров от компании Infragistics. Available at: https://pvs-studio.ru/ru/blog/posts/csharp/0375/, accessed: 22-05-2025.

[15]. J. Samhi, R. Just, T. F. Bissyand´e, M. D. Ernst и J. Klein. Call Graph Soundness in Android Static Analysis. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 945-957, 2024.

[16]. L. Sui, J. Dietrich, M. Emery, S. Rasheed и A. Tahir. On the soundness of call graph construction in the presence of dynamic language features – a benchmark and tool evaluation. In Programming Languages and Systems: 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings 16, pp. 69-88. Springer, 2018.

[17]. M. Khedkar. Call Graph Construction for Spring Framework. Дис. док., CHENNAI MATHEMATICAL INSTITUTE, 2020.

[18]. S. Yang, D. Yan, H. Wu, Y. Wang и A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, том 1, pp. 89-99. IEEE, 2015.

[19]. C. Sells и I. Griffiths. Programming WPF: Building Windows UI with Windows Presentation Foundation. O'Reilly Media, Inc., 2007.

[20]. XAML language overview – WPF | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/, accessed: 22-05-2025.

[21]. 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 / Koshelev V.K., Ignatyev V.N., Borzilov A.I. C# static analysis framework. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 1, 2016, pp. 21-40 (in Russian). DOI: 10.15514/ISPRAS-2016-28(1)-2.

[22]. В. К. Кошелев, И. А. Дудина, В. Игнатьев и А. И. Борзилов. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 59-86. / Koshelev V., Dudina I., Ignatyev V., Borzilov A. Path-sensitive bug detection analysis of C# program illustrated by null pointer dereference. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). 2015;27(5):59-86. (In Russ.) DOI: 10.15514/ISPRAS-2015-27(5)-5

[23]. How MSBuild builds projects - MSBuild | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/visualstudio/msbuild/build-process-overview, accessed: 22-05-2025.

[24]. Chordious. Available at: https://chordious.com/, accessed: 22-05-2025.

[25]. ScreenToGif. Available at: https://www.screentogif.com/, accessed: 22-05-2025.

[26]. JPPhotoManager. Available at: https://github.com/jpablodrexler/jp-photo-manager, accessed: 22-05-2025.

[27]. ratishphilip/nvmsharp: A WPF application which provides robust features to manage Environment Variables in a Windows desktop. Available at: https://github.com/ratishphilip/nvmsharp, accessed: 22-05-2025.

[28]. 0x78654C/PwM: Simple password manager in C# WPF to store locally sensitive authentication data for a specific application. Available at: https://github.com/0x78654C/PwM, accessed 22-05-2025.

Информация об авторах / Information about authors

Никита Александрович БОРОДАВКО – студент бакалавриата факультета вычислительной математики и кибернетики МГУ, сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ исходного кода, символьное выполнение, поиск дефектов в исходном коде.

Nikita Alexandrovich BORODAVKO is a bachelor's student at the Department of Computational Mathematics and Cybernetics of Lomonosov Moscow State University, an employee of the ISP RAS. Research interests: compiler technologies, static analysis of the source code, symbolic execution, finding errors 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.