Инфраструктура статического анализа программ на языке C#

В. К. Кошелев <vedun@ispras.ru>
В. Н. Игнатьев <valery.ignatyev@ispras.ru>
А. И. Борзилов <helendile@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25.

Аннотация. В работе рассмотрены различные аспекты статического анализа программ на языке C# с целью обнаружения максимального количества ошибок за минимально приемлемое время. Описан полный цикл статического анализа программного обеспечения, при этом основное внимание уделяется особенностям, возникающим при анализе языка C#. Рассмотрены методы, позволяющие учитывать популярные возможности языка на всех уровнях анализа: построения графа вызовов и графа потока управления, проведение анализа потоков данных и чувствительного к контексту и путям межпроцедурного анализа. Предлагается метод символьного исполнения, основанный на таких работах, как Bounded Model Checking и Saturn Software Analysis Project. В статье описана организация модели памяти, позволяющая как проводить точный внутрипроцедурный анализ, так и создавать компактные представления для привязанных к функциям условий, используемые при межпроцедурном анализе. Особое внимание уделяется теме оптимизации возникающих на этапе чувствительного к путям анализа условий. Условия необходимо оптимизировать как по размеру, поскольку при межпроцедурном чувствительном к путям анализе необходимо сохранять большое количество условий для каждой проанализированной функции, так и по сложности, поскольку время анализа ограничено. Решение условий производится при помощи современных SMT-решателей, таких как Microsoft Z3 Prover. В статье также рассмотрены различные подходы к моделированию поведения библиотечных функций: при помощи резюме в виде набора признаком или в виде упрощенных реализаций на языке C#. Все приведённые решения реализованы в инструменте статического анализа SharpChecker и протестированы на наборе проектов различного объема (от 1.5 тыс. до 1.35 млн. строк кода) с открытым исходным кодом.

Ключевые слова: Статический анализ; использование нулевого указателя; чувствительность к путям; чувствительность к контексту; резюме функции; поиск дефектов; Roslyn; C#

DOI: 10.15514/ISPRAS-2016-28(1)-2

Для цитирования: 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

1. Введение

Статический анализ программ в последние годы занимает важную роль при разработке программного обеспечения, позволяя обнаруживать ошибки в ПО без фактического исполнения программы. За счёт использования все более сложных методов анализа появляется возможность повысить качество результатов, обнаруживая новые ошибки и снижая относительное количество ложных срабатываний.

В настоящее время язык программирования C# достиг высокой популярности и, согласно индексу TIOBE [1], занимает четвертую позицию в рейтинге распространённости. Благодаря хорошей архитектуре, богатому выбору доступных библиотек и хорошо поддерживаемой инфраструктуре программирования все больше промышленного и открытого ПО разрабатывается с использованием этого языка. Это означает, что к программам предъявляются высокие требования по качеству как результирующей программы, так и исходного текста. Для соответствия таким стандартам качества существуют различные методы, одним из которых является использование инструментов статического анализа исходного текста. Учитывая тот факт, что C# не используется при разработке низкоуровневого ПО, а большая часть существующих проектов на C# имеет существенно больший размер, чем драйверы операционной системы или встраиваемое в аппаратуру ПО, задача формальной верификации с помощью статического анализа не является актуальной для C#. В то же время, задача обнаружения максимального количества ошибок за минимально возможное время является очень востребованной.

Специфика разрабатываемых на рассматриваемом языке проектов определяет список наиболее важных типов ошибок, которые могут приводить к отказам ПО или даже позволят злоумышленнику эксплуатировать найденные уязвимости.

Основные проблемы могут возникать из-за:

- использования null;
- утечки ресурсов (памяти или, например, файловых дескрипторов), полученных при взаимодействии с unmanaged кодом;
- использования уже освобождённых unmanaged объектов;
- попадания пользовательских данных в базы данных или пользовательский интерфейс без корректной проверки (SQL Injection, XSS и т.д.)
- ошибок, связанных с явным приведением типов.

Как несложно заметить, список важных проблем отличается от соответствующего списка для языков C/C++. Например, очень важной проблемой для C++ является возможность доступа к данным за пределами отведённого буфера, которая не является актуальной для C# по двум причинам. Во-первых, при доступе за границы массива возникнет

соответствующее исключения, и эксплуатировать такую уязвимость с целью манипуляции чужими данными невозможно. Во-вторых, наличие встроенных в язык средств работы с коллекциями привело к тому, что доступ по индексу практически не используется в реальном ПО, а реализуется с помощью итераторов и Linq [2].

Большинство разработанных на данный момент инструментов статического анализа создавались для языков C/C++ и Java и не могут без серьёзных изменений хорошо работать с программами на C#. Постоянное использование исключений, делегатов, Linq, вызовов через интерфейсы, свойств классов, которые автоматически вызывают геттеры и сеттеры, концепция Disposable объектов для взаимодействия с unmanaged окружением, встроенные синтаксические конструкции и одновременно библиотечные методы для организации блокировок не позволяют без существенных доработок применять существующие инструменты анализа. Поэтому в данной работе подробно рассмотрены практические приёмы, позволяющие учитывать особенности языка на всех уровнях анализа: начиная с построения графа потока управления и графа вызовов, заканчивая анализом условий для достижения чувствительности к контексту вызова и путям выполнения.

Существующие инструменты статического анализа C# промышленного уровня можно разделить на две группы: поддерживающие контекстно- и потоково-чувствительный анализ (разрабатываемый в ИСП РАН инструмент Svace [3], инструменты от Coverity [4], Klocwork [5]), и основанные на абстрактном синтаксическом дереве (АСД) (инструменты от SonarLint [6], СиПроВер [7]). Продукты из первой группы имеют очень высокую стоимость и применяются в основном в крупных компаниях, а легковесные анализаторы из второй группы гораздо более доступны, однако принципиально не могут обнаружить большой класс ошибок.

Приведённые аргументы свидетельствуют о необходимости адаптации алгоритмов статического анализа, созданных для C++ и Java, для использования в анализе C#. Кроме того, задача разработки контексто- и потоково- чувствительного инструмента статического анализа является очень актуальной, поскольку на данный момент практически отсутствуют доступные аналогичные инструменты.

Статья организована следующим образом. В части 2 показана общая схема проведения анализа, которая в соответствии с этапами анализа уточняется в последующих частях. Так, в 3 части рассмотрены вопросы построения графа вызовов, в 4 – особенности графа потока управления. Части 5-7 описывают схему организации чувствительного к путям и контексту межпроцедурного анализа. В восьмом разделе приведена оценка результатов тестирования инструмента SharpChecker [8], реализующего все рассмотренные подходы. В заключении подведены итоги работы.

2. Схема работы анализатора

Разработанный инструмент статического анализа SharpChecker, описанный в данной работе, способен находить 30 различных типов ошибок, среди которых есть как использующие исключительно синтаксический анализ, так и анализ потоков данных, а также наиболее мощный чувствительный к контексту и путям выполнения межпроцедурный анализ.

Для понимания особенностей обработки специфичных возможностей языка C# предварительно рассмотрим общую схему работы анализатора, представленную на рисунке 1.

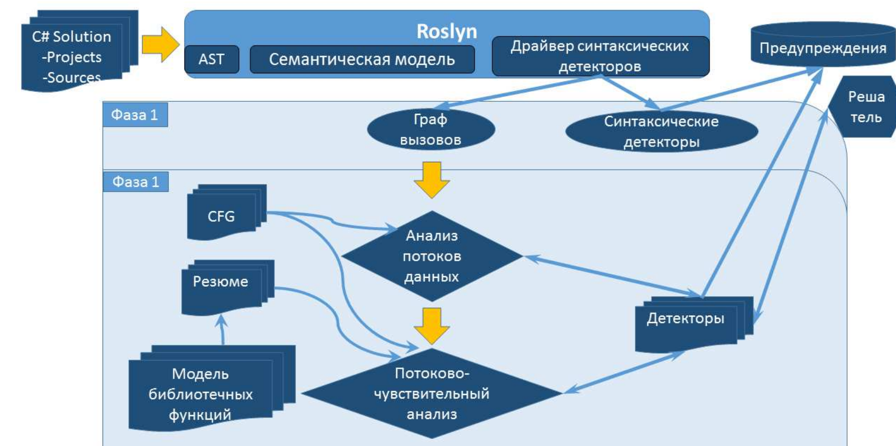

Рис. 1. Схема работы анализатора SharpChecker
Fig. 1. Operational scheme of the SharpChecker analyzer

Большинство проектов на языке C# используют для организации сборки механизмы, основанные на файлах проектов и решений, предоставляемые Microsoft Visual Studio [9]. Разработанный анализатор использует инфраструктуру Roslyn [10] для работы с файлами системы сборки, а также для компиляции исходного текста программы. Roslyn – это открытая компиляторная платформа, поддерживающая набор механизмов для разработки статических анализаторов. Инструмент SharpChecker использует только часть Roslyn, отвечающую за разбор файлов проектов, решений, в результате которой определяется множество файлов, используемое для сборки заданной программы или библиотеки, а также необходимое окружение. Кроме этого, Roslyn использован для построения абстрактного синтаксического дерева и таблицы символов каждого модуля компиляции. Для достижения хорошего качества результатов статический анализатор обязан учитывать правила сборки, определяющие, в первую очередь, правила для компоновщика, или, другими словами, граф вызовов. Таким образом, на

первом этапе с помощью Roslyn выполняется построение абстрактного синтаксического дерева для всех файлов, участвующих в сборке, производится анализ всех возможных вызовов, в том числе неявных, строится иерархия наследования классов и собирается другая информация, необходимая для построения статического графа вызовов. Одновременно, на этом же этапе выполняется поиск синтаксических ошибок, для обнаружения которых достаточно АСД. Второй этап реализует построение графа вызовов программы на основе собранных данных, а затем уточняет его на основе результатов анализа виртуальных функций и интерфейсов. Особенности построения графа вызовов подробнее описаны в части 3.

Третий этап состоит в обходе графа вызовов в обратном топологическом порядке от вызываемой функции к вызывающей. При этом возможные циклы в графе вызовов разрываются в произвольном месте. В течение этой стадии анализа выполняются как анализ потоков данных и использующие его детекторы, так и потоково-чувствительные детекторы. Кроме того, часть собранной информации сохраняется в резюме функции и используется при анализе функций, вызывающих данную. Обход функций производится параллельно, что позволяет существенно сократить время работы на многоядерных машинах.

Следующий этап реализует проверку большинства правил, а также накопление информации о результатах анализа функций в резюме для последующего использования в вызывающей функции.

Таким образом, в процессе обхода графа вызовов по предоставленным Roslyn АСД и таблице символов осуществляется построение графа потока управления (ГПУ) для каждой функции. При этом анализатор использует два представления ГПУ для каждой функции, различающиеся обработкой лямбда-функций. Особенности построения ГПУ для специфичных конструкций языка C# рассмотрены в части 4.

Следующий этап реализует классический анализ потоков данных, на основе которого работают детекторы, например, поиск неиспользованных значений, включающий как неиспользуемые переменные, так и результаты вычислений.

При этом происходит обход базовых блоков ГПУ, во время которого детекторы вычисляют свои множества, соответствующие GEN и KILL [11] в терминах классического анализа, а затем по правилам, заданным в тех же детекторах, обходится граф потока управления для построения IN и OUT. Наконец, вызывается обработчик завершения анализа, который использует собранную информацию для выдачи предупреждений.

После этапа анализа потоков данных работает чувствительный к путям анализ.

Здесь выполняется накопление информации в резюме каждой функции и анализ всех возможных путей выполнения программы с учётом уже построенных резюме.

Описанная схема работы позволяет эффективно сочетать поиск различных типов ошибок в программе, постепенно накапливая недостающую информацию и освобождая ненужные данные.

3. Построение графа вызовов

В рассматриваемом инструменте граф вызовов необходим для решения двух основных задач.

1. Для каждого вызова в программе (с учётом делегатов, интерфейсов, геттеров и сеттеров свойств, лямбда-выражений) определить вызываемую функцию или множество функций, которые могут быть вызваны.
2. Построить правильный порядок обхода функций, чтобы вызываемые функции были проанализированы к моменту анализа вызывающих. Это позволяет достичь контекстной чувствительности за счёт использования резюме вызываемых функций.

Построение графа вызовов состоит из двух этапов. Сначала осуществляется обход АСД, во время которого происходит:

1. Анализ вершин, соответствующих объявлениям класса, для построения иерархии классов. Эта информация используется на втором этапе для уточнения графа в контексте виртуальных вызовов.
2. Анализ вершин АСД, в которых возможен вызов:
    a. непосредственный вызов метода, статического метода, конструктора;
    b. доступ к свойству объекта, когда возможен вызов геттера или сеттера;
    c. создание лямбда-выражений и анонимных функций;
    d. неявные вызовы переопределённых операторов;
    e. неявные вызовы операторов преобразования;
    f. вызовы финализаторов (деструкторов).

    Для каждого узла АСД, в котором происходит вызов, осуществляется связывание по строковому идентификатору функции с существующим резюме или создаётся новое резюме.

После завершения компиляции всего решения выполняется построение графа вызовов при помощи адаптированного для C# алгоритма Class Hierarchy Analysis (CHA) [12]. Полученный в результате граф топологически сортируется, после чего производится выбор функций, допускающих параллельный анализ.

4. Построение графа потока управления

Граф потока управления лежит в основе анализа и должен отражать все возможные пути выполнения программы. В рассматриваемом анализаторе

базовые блоки состоят из вершин АСД. Это позволяет сохранить связь между сложными анализами путей исполнения и соответствующими оригинальными конструкциями языка. Использование языково-независимого представления, например, трехадресного кода, позволило бы упростить анализ за счет избавления от синтаксического сахара. Однако такой подход приводит к потере информации об исходных конструкциях, что влияет на качество последующего анализа и, особенно, содержания предупреждений. В примере 1 показан единственный содержательный базовый блок ГПУ для метода `foo`. Каждая инструкция базового блока – это вершина АСД, тип которой указан в правом столбце. Конструкция **[число]** означает номер инструкции в базовом блоке, результат которой используется в качестве аргумента в данной инструкции.

```
public void foo()          0: 1                      LiteralExpression
{                          1: int p = [0]          VariableDeclaration
   int p = 1;              2: MyClass                   IdentifierName
   bar(p,                  3: new [2]()     ObjectCreationExpression
     new MyClass());       4: p                         IdentifierName
}                          5: bar                       IdentifierName
                           6: [5]([4], [3])     InvocationExpression
```

Пример 1. Базовый блок графа потока управления
Example 1. Basic block of control flow graph

Ребра, соединяющие базовые блоки, хранят разные атрибуты, указывающие на то, является ли ребро обратным для циклов, соответствует положительной или отрицательной ветви условного оператора, является ли входом, выходом или замыканием лямбда-выражения, соответствует ли ребро переходу по пользовательскому или системному, явному или неявному исключению и некоторые другие. Кроме того, на рёбрах хранятся условия перехода для чувствительного к путям анализа.

Для упрощения работы с циклами на фазе чувствительного к путям анализа, условие входа в цикл дублируется в его конце, чтобы позволить совершить выход из цикла без прохода по обратному ребру.

Существенное влияние на результаты анализа оказывает способ представления исключений в ГПУ. В языке C# практически каждая инструкция в исходном коде может вызвать исключение – например, NullReferenceException. Практика программирования также поощряет использование исключений. В итоге вызов практически любой функции может закончиться выбросом исключения. Практические эксперименты показали, что необходимо разделять пользовательские и системные исключения, выброшенные явно оператором throw и пришедшие из вызовов библиотечных функций. Такое разделение используется в детекторе утечки unmanaged памяти в случае, например, отсутствия свободного места на диске

или ошибки передачи данных по сети. Кроме того, для корректной обработки явных перехватов исключений, требуется знать, какие исключения могут возникнуть во время выполнения функции.

Таким образом, при построении ГПУ для каждого вызова функции необходим список всех возможных в ней исключений. Для пользовательских функций этот список строится во время анализа, а информация о возможных исключениях при работе библиотечных функций загружается из базы данных резюме, более подробное описание которой содержится в части 5.

Разбиение базового блока и создание ребра для исключения после каждого вызова существенно увеличивает как сложность самого графа, так и время его последующей обработки. Поэтому на данный момент дробление базового блока осуществляется только для явных пользовательских исключений. Остальные добавляются как возможные выходы в конце базового блока. Однако это решение имеет недостатки. Рассмотрим пример сравнительно частой ситуации.

```
public StreamReader GetStreamReaderOrThrow(bool b)
{
     if (b)
         return new StreamReader("stream.txt");
     throw new NotImplementedException();
 }

 public void Test(bool b)
 {
     StreamReader reader = null;
     try
     {
         reader = GetStreamReaderOrThrow(b);
     }
     finally
     {
         reader.Dispose();
     }
 }
```

Пример 2. Код на C#, содержащий возможное использование null
Example 2. C #-code with possible use of null

Если в функции `Test` не добавлять ребро между точкой вызова и присваиванием, то ошибку возможного использования *null* в `reader.Dispose()` найти не получится.

Широкое распространение в коде на C# имеют также лямбда-выражения, лямбда-функции и анонимные функции. Для упрощения будем называть все

эти сущности одним словом лямбды. Без точного межпроцедурного чувствительного к путям анализа определить место их вызова невозможно. Поэтому для нужд различных детекторов используются различные эвристики. На этапе анализа потоков данных работают сравнительно простые детекторы, поэтому для них разумным компромиссом является непосредственное встраивание тела лямбды в ГПУ. Поскольку количество исполнений лямбды также неизвестно, добавляются ребро, минующее тело лямбды, и ребро, ведущее из ее выхода во вход. Такое встраивание позволяет повысить качество анализа по аналогии с компиляторной оптимизацией inlining, давая анализу информацию о том, что будет происходить с переменными, использованными в лямбдах. Например, это позволяет существенно повысить качество анализа неиспользуемых значений без более слабых эвристик, считающих, что лямбды используют все переменные. По окончанию анализа потоков данных ребра, соединяющие тело лямбды с остальной функцией, разрываются. Таким образом, в дальнейшем лямбды не учитываются.

Язык C# содержит достаточное количество удобного синтаксического сахара, который также требуется представлять в ГПУ. К таким конструкциям относятся операторы “??”, “?.”, yield break, yield return, switch по строкам и goto по вычисляемым выражениям, интерполированные строки и т.д. Основная сложность их поддержки в графе потока управления заключается в аккуратном создании базовых блоков и рёбер между ними, а также построении правильных условий переходов по этим рёбрам на этапе анализа путей.

Таким образом, построение ГПУ C# программы для последующего использования в статическом анализаторе – это поиск компромиссов между сложностью предшествующего построению анализа и последующего. Набор предложенных выше методов построения сохраняет большую часть информации в удобном для последующего обхода и анализа виде. Обходчик путей выполнения программы по ГПУ выдаёт набор предопределённых событий, обработчики которых реализованы в детекторах.

5. Чувствительный к путям анализ

В отличие от большинства доступных инструментов статического анализа, SharpChecker поддерживает чувствительный к путям и контексту анализ. Такой анализ позволяет обнаруживать наиболее интересные типы ошибок, такие как утечка ресурсов, использование *null*, ошибки приведения типов.

Для проведения внутрипроцедурного анализа используется подход, схожий с подходом BMC[13]. Подход BMC рассматривает пути на графе потока управления, которые начинаются в точке входа в программу, и длина которых не превышает некоторую константу. При подсчёте длины допускается наличие разных весов у разных рёбер графа. В данной работе для выполнения развёртки циклов считается, что прямое ребро ГПУ имеет вес ноль, а обратное – один. Таким образом в ходе анализа рассматривается лишь конечный набор

путей в ГПУ. В отличие от оригинального BMC, рассматриваемый подход при обработке вызовов функций использует резюме функций вместе вставки, а также использует объединение состояний в точках слияния.

Для представления множества путей ГПУ, которые будут проанализированы, удобно ввести понятия графа развёртки. Графом развёртки будем называть граф, обладающий следующими свойствами:

- граф развертки является ациклическим;
- каждая вершина графа развертки сопоставлена какой-либо вершине ГПУ;
- в графе развертки выделены две вершины: входная и выходная, которые сопоставлены входной и выходной вершине графа ГПУ соответственно;
- между двумя вершинами графа развертки есть ребро только в том случае, если в ГПУ есть ребро между соответствующими вершинами;
- все вершины графа развертки достижимы из входной вершины;
- выходная вершина достижима из всех вершин графа развертки;

Каждому пути в графе развертки по построению соответствует путь в ГПУ, следовательно, граф развертки задает множество путей в ГПУ. В том случае, если какая-либо вершина в графе развертки имеет более одного входного ребра, будем считать, что для последующего анализа инструкций данной вершины необходимо вначале произвести объединение входных состояний. Таким образом граф развертки задает не только множество рассматриваемых путей, но и набор вершин, в которых будет производиться объединение состояний.

Далее будем считать, что граф развертки уже построен. В данной работе используется алгоритм построения графа развертки, описанный в статье [14]. Весь дальнейший анализ проводится не для ГПУ, а для графа развертки, что корректно ввиду соответствия их путей.

В данном подходе точкой входа анализа является точка входа в анализируемую функцию. Начальное состояние в точке входа, включающее аргументы функции и состояние памяти, параметризуется набором типизированных символьных переменных. Для параметризации аргументов функции достаточно рассмотреть отображение $Params : P \to S$, где P это множество аргументов функции, а S – множество символьных переменных. Будем различать три типа символьных переменных: целочисленную, булевую и ссылочную – соответственно, S_I, S_B и S_R. Все целочисленные символьные переменные являются знаковыми с фиксированным размером в битах. Целочисленные и булевые символьные переменные могут принимать любое допустимое их типом значение. Значения ссылочных переменных удовлетворяют следующему ограничению: две ссылочные переменные равны в том и только том случае, если они обе примут значение null. Из этого следует, что никакие ссылочные переменные не являются алиасами.

Кроме явно заданных параметров, начальное состояние также задает состояние памяти. Для описания параметризованного состояния памяти введем частичное отображение $Heap : S_R \times F \rightharpoonup S$, которое паре «символьная переменная и поле» ставит в соответствие символьную переменную, полученную при соответствующем чтении из поля. Параметризованным начальным состоянием будем называть пару отображений $State_{entry} = \langle Params, Heap \rangle$.

Для задания $Params$ достаточно каждому параметру функции сопоставить уникальную символьную переменную. Задание $Heap$ осуществляется лениво. В том случае, когда к переменной, содержащей в качестве значения символьную переменную s, применяется операция чтения поля f, и пара $\langle s, f \rangle$ не содерждится в Heap, Heap доопределяется $\langle s, f \rangle \to s_{new}$, где s_{new} – новая символьная переменная. В силу того, что анализу подвергается только конечный набор путей, число операций с памятью будет тоже конечным, следовательно, отображение Heap также будет определено на конечном наборе пар. Здесь и далее под состоянием памяти понимается состояние лишь той части памяти, которая доступна с помощью операций работы с памятью на рассматриваемых путях выполнения.

Параметризированное начальное состояние позволяет представлять результаты выполнения операторов функции в виде выражений над символьными переменными. Каждое символьное выражение также имеет один из трех типов: целочисленный, булевый или ссылочный. Правила построения символьных выражений следующие:

- константы являются символьными выражениями соответствующих типов;
- символьные переменные являются символьными выражениями соответствующих типов;
- для двух символьных выражений одинаковых типов определены операции сравнения, которые являются булевыми символьными выражениями;
- для двух целочисленных символьных выражений определены все арифметические и битовые операции, которые также являются целочисленными символьными выражениями;
- для двух булевых символьных выражений определены все стандартные булевы операции, которые также являются булевыми символьными выражениями;
- для символьных выражений определены стандартные унарные операции, которые также являются символьными выражениями соответствующих типов;
- для двух символьных выражений одинаковых типов и булевого символьного выражения определена тернарная условная операция того же типа, результат которой совпадает с первых выражением, если булевое выражение верно, а иначе совпадает со вторым выражением.

Множество построенных таким образом символьных выражений обозначим как SE. Отдельно будем выделять SE_B.и SE_R, булевое символьное выражение и ссылочное символьное выражение соответственно.

Для задания резюме функции необходимо ввести два дополнительных отображения: $Param_{out} : P \to SE$ и $Heap_{out} : S_R \times F \rightharpoonup SE$. Отображение $Param_{out}$ задает соответствие между выходными параметрами функции и символьными переменными начального состояния, а $Heap_{out}$ между состоянием памяти после выполнения функции и символьными переменными. Тогда резюме функции задает отображения $State_{entry}$, $Param_{out}$,$Heap_{out}$. Отображения $Param_{out}$ и $Heap_{out}$ могут содержать дополнительные символьные переменные, не являющиеся частью параметризированного входного состояния. Наличие данных переменных обусловлено как наличием вызовов функций, реализация которых отсутствует, так и ограничением на размер резюме. В том случае, если число различных символьных выражений в $Param_{out}$ и $Heap_{out}$ чрезмерно велико, часть из них заменяется на новые неизвестные символьные переменные. Новые символьные переменные, возникшие при применении резюме, также становятся частью параметризации начального состояния.

Пусть при анализе конкретной функции построено её параметризованное начальное состояние, а также резюме всех вызываемых в ней функций. Тогда абстрактным состоянием для заданного ребра графа развертки назовем множество возможных состояний памяти и переменных относительно выбранной параметризации начального состояния. Начальное состояние может быть описано тройкой

$$State = \langle Var: V \to SE, Heap: S \times F \to SE,\ Pred \rangle, Pred \in SE_B.$$

Var и $Heap$ задают состояние переменных и памяти через символьные переменные, а $Pred$ является предикатом пути, задающим ограничения на символьные переменные. Тогда каждое решение предиката пути, т.е. такой набор символьных переменных на котором $Pred$ обращается в истину, будет однозначно задавать состояние переменных и памяти.

В работе [14] представлен метод, позволяющий по заданным графу развертки и набору резюме вызываемых функций строить абстрактные состояния $State$ для каждого ребра графа развертки. В данном методе, резюме для анализируемой функции строится из абстрактного состояния $State$, полученного в выходной вершине графа развертки.

Построение детекторов ошибок происходит на базе абстрактного состояния $State$. Каждый детектор может расширить $State$, добавив к нему дополнительную информацию и определив правила объединения этой информации. Типичным примером дополнительной информации является частичное отображение $SE \rightharpoonup SE_B$. Такое отображение задает условие, при

котором символьное выражение обладает заданным свойством. Примерами таких свойств являются «символьное выражение сравнивалось с явным null» или «символьное выражение является ссылкой на освобожденный ресурс». Детектор ошибки может следить за символьным выполнением, изменяя свою часть абстрактного состояния. В случае, если символьное выражение, обладающее заданным свойством при данном условии, используется в потенциально опасной операции, то детектор выдаст предупреждение, если предикат пути и данное условие имеют общее решение.

6. Вычисление условий

Провести вычисления предиката пути можно наивным алгоритмом, пересчитывая предикат с учетом условий на ребрах и объединяя предикаты путей различных абстрактных состояний в точках объединения. Данный алгоритм предложен в предыдущей работе [14]. Однако в силу того, что объединению зачастую подвергаются ветки оператора if с противоположными условиями, итоговые формулы предикатов путей могут быть сильно упрощены. Такое упрощение используется как для уменьшения потребления памяти при сохранении предикатов в резюме, так и для ускорения решения совместности формул.

Для упрощения формул можно использовать два подхода. Первый подход заключается в использовании специального представления, автоматически упрощающего формулу во время её построения. Примером такого представления являются ROBBD [15].

Второй подход заключается в использовании свойства графа развертки: если через две вершины проходят одни и те же пути, то и предикаты путей в этих вершинах будут совпадать. Рассмотрим его подробнее.

Выберем конкретную вершину в графе развертки. Будем считать, что все вершины топологически меньше данной уже обработаны и их предикаты пути посчитаны. Найдем для данной вершины её область пост-доминирования. Рассмотрим граф вершин топологически меньших либо равных данной, назовём его G'. Построим разрез $\langle S, T\rangle$, такой, что к S относятся все вершины G', которые в исходном графе не пост-доминирует данная вершина, а к T – все вершины из области пост-доминирования. Пусть $\{s_i, t_i\}$ – список ребер, лежащих на разрезе. Тогда предикат пути для данной вершины построим следующим образом: $\bigvee_i (Pred(s_i) \wedge Cond(s_i, t_i))$, где $Pred(s_i)$ – предикат пути для вершины s_i, а $Cond(s_i, t_i)$ – условие перехода по ребру.

Стоит отметить, что оба подхода к минимизации размера формул могут применяться одновременно.

Перейдем к вычислению условия объединения. Пусть дана тройка вершин $\langle lhs, rhs, join\rangle$, таких, что есть ребра $\langle lhs, join\rangle$ и $\langle rhs, join\rangle$. Заранее известно, что вершина *join* была достигнута либо на пути, прошедшем через ребро $\langle lhs, join\rangle$, либо на пути, прошедшем через ребро $\langle rhs, join\rangle$. Для того, чтобы

различать, по какому из этих двух ребер была достигнута вершина *join*, достаточно воспользоваться условием $lhs_{cond} = Pred(lhs) \vee Cond(lhs, join)$, соответствующим тому, что путь прошел по ребру $\langle lhs, join\rangle$. Аналогично, можно рассмотреть обратное условие $rhs_{cond} = Pred(rhs) \vee Cond(rhs, join)$ для ребра $\langle rhs, join\rangle$. В силу детерминированности выбранной параметризации, условия lhs_{cond} и rhs_{cond} несовместны. Однако эти условия зачастую слишком громоздкие, и для хорошей производительности их необходимо упростить.

Задачу упрощения можно сформулировать следующим образом. Необходимо найти такое условие *Interpol*, что $Interpol \rightarrow lhs_{cond}$ и $\neg Interpol \rightarrow rhs_{cond}$. Такое условие можно получить, применив к формулам lhs_{cond} и rhs_{cond} интерполирующий решатель [16]. Применение такого решателя на каждую операцию объединения достаточно затратно, поэтому рассмотрим алгоритм построения условия *Interpol* на основе дерева доминаторов.

Пусть dom это непосредственный доминатор вершин lhs и rhs, тогда рассмотрим множество вершин топологически меньше либо равных lhs и больше либо равных dom, назовём его L. Рассмотрим аналогичное множество для rhs, назовем его R. Рассмотрим тогда $L \cap R$ и $L \setminus R$, без ограничения общности будем считать, что $L \setminus R$ не пусто. Тогда $Interpol = \bigvee_i (Pred_{dom}(u_i) \wedge Cond(u_i, v_i))$, где $\{u_i, v_i\}$ – ребра, лежащие на разрезе $\langle L \cap R, L \setminus R\rangle$, а $Pred_{dom}(u)$ предикат пути из вершины dom до вершины u.

7. Резюме внешних функций

С точки зрения анализатора функции делятся на пользовательские, т.е. те, для которых есть исходный код, и внешние, или библиотечные, – из системных или пользовательских библиотек. В связи с этим, их резюме существенно отличаются.

Рассмотрим сначала, где можно взять информацию о внешних функциях. Во-первых, их можно анализировать в бинарном виде. Поскольку большинство распространённых библиотек для C# скомпилировано в CIL [17], можно использовать, например, Mono.Cecil[18] для дополнительного анализа бинарного представления. Одна из первых версий описываемого инструмента использовала этот подход, однако впоследствии решено было отказаться от анализа CIL, поскольку для получения качественных результатов необходима серьёзная инфраструктура, практически дублирующая функциональность анализатора исходного кода, что требует существенного времени на реализацию.

Поскольку большинство исходных текстов для популярных библиотек доступно для скачивания и анализа, другим способом является предварительный анализ исходного кода библиотеки по аналогии с пользовательскими функциями или даже анализ на лету. Второй способ

требует распространения огромного количества исходного кода библиотек вместе с инструментом, а также существенно увеличивает объем анализируемого кода, поскольку многие функции имеют сложную реализацию, кроме того используют другие функции без исходного текста, включая unmanaged библиотеки, например, WinAPI [19].

Предварительный анализ также не решает проблему с функциями без исходного кода, а также требует разработки механизмов сериализации и постоянного обновления после исправления ошибок или улучшения качества анализатора. Кроме того, для корректного анализа требуется поддержка различных версий одних и тех же библиотек.

Опыт разработки показывает, что для анализа библиотечных функций в большинстве случаев достаточно информации, представимой в виде нескольких десятков текстовых или логических атрибутов. Поэтому одним из используемых на данный момент решений является SQL база данных, содержащая записи для более 60000 популярных функций. База создавалась путём автоматизированного разбора и анализа официальной документации MSDN [20], а также исправлений и дополнений по результатам оценки работы инструмента. Для каждой функции хранится битовый вектор, представляющий логические свойства, например, что она возвращает Disposable объект и различные другие свойства, например, список возможных исключений. Подобным образом хранится информация о параметрах, например, что они не могут быть *null*. Такой подход позволяет быстро, достаточно качественно и предсказуемо предоставлять информацию, достаточную для большинства детекторов анализатора.

Серьёзным недостатком данного подхода является сложность моделирования сторонних эффектов и условий их возникновения. Рассмотрим характерный пример 3.

```
public string Foo()
{
    var list = new List<int>();
    list.Add(5);
    string check = null;
    foreach (var elem in list)
    {
        check = "Not null";
    }
    return check.ToString();
}
```

Пример 3. Код на C#, порождающий ложное срабатывание без дополнительного моделирования библиотечных функций
Example 3. C #-code generating a false alarm without additional modeling library functions

В результате добавления в `list` элемента 5 список `list` перестает быть пустым, поэтому тело цикла `foreach` всегда выполняется, и переменная `check` всегда инициализируется, следовательно, использование *null* невозможно. Однако анализатор предполагает, что `list` может быть пустым, потому что в описанной модели функции `List<T>.Add(T)` нет возможности прямо указывать подобные эффекты. Поэтому переход по ребру в обход `foreach` возможен, что приводит к выдаче ложного предупреждения.

Для поддержки сложных эффектов используется моделирование внешних функций на языке C#. Модель функции в данном случае представляет собой упрощённую реализацию. Такая реализация компилируется и анализируется на лету как пользовательская функция, и для неё строится резюме, как для функции, имеющей исходный код.

8. Результаты работы инструмента

В данном разделе приводятся результаты тестировании инструмента SharpChecker на наборе проектов с открытым исходным кодом. В таблице 1 представлено количество предупреждений, выданных для различных проектов, а также время работы анализатора. Представлены наиболее интересные с точки зрения данной работы группы ошибок, при поиске которых используется чувствительный к путям и контексту анализ:

- Null reference – ошибки, связанные с потенциальным использованием null. В данную группу объединены как использование переменной, которой могло быть явно присвоено значение null, так и использование переменной после или до сравнения с null;
- Resource leak – утечки ресурсов, связанные с отсутствием Dispose или неправильным использованием конструкции using;
- Wrong cast – ошибки приведения типов;
- Unreachable code – недостижимый код.

Таблица 1. Результаты тестирования инструмента SharpChecker на наборе программ с открытым исходным кодом
Table 1. Results of the Sharp Checker tool testing on a set of open source software

Проект	LOC	Null reference	Resource leak	Wrong cast	Unreachable code	Время анализа (мин:сек)
Sake	1K	0	0	0	0	0:12
Polly	5K	0	0	0	2	0:21
BobBuilder	6.5K	1	14	0	3	0:12
Shadowsocks	18K	0	10	4	6	0:42

Perspective	20K	10	1	0	1	0:20
CSParser	21K	13	1	0	7	0:48
NetMQ	30K	11	10	1	7	0:47
Jil	49K	33	1	0	20	8:14
LibGit	51K	15	14	0	1	1:06
OpenBVE	57K	5	18	0	50	14:03
Cassandra	63K	19	53	2	4	1:44
OpenRA	105K	42	31	0	22	8:11
FSpot	110K	179	24	0	30	1:47
ShareX	145K	42	53	0	23	2:45
Banshee	168K	143	20	0	34	3:03
Lucene.Net	528K	63	820	1	87	13:39
Roslyn	1.35M	1399	0	5	139	36:59

Можно заметить, что объем проекта не всегда коррелирует с временем анализа: это объясняется наличием в некоторых проектах сложных участков кода. Большое количество NRE на некоторых проектах объясняется стилем кодирования, при котором оператор *as* используется вместо явного приведения типа. Большое количество утечек ресурсов происходит в тестах.

9. Заключение

Разработка инструментов статического анализа – это поиск компромисса между скоростью работы и качеством результата. В отличие от компилятора, статический анализатор допускает некоторые эвристики, позволяющие ускорить работу за счёт неконсервативной обработки наиболее ресурсоёмких конструкций или процессов. В работе представлен набор методов, в том числе эвристических, комбинация которых позволяет достигать хороших результатов как по качеству результата, так и по времени работы.

На базе компиляторной инфраструктуры Roslyn разработан инструмент статического анализа программ на языке C# SharpChecker [8]. В данном инструменте реализованы предложенные методы организации чувствительного к путям и контексту межпроцедурного анализа. Результаты тестирования инструмента SharpChecker показывают его высокую эффективность на промышленных проектах с открытым исходным кодом.

Список литературы

[1]. Веб-сайт TIOBE Index [HTML] http://www.tiobe.com/tiobe_index

[2]. Описание инструмента LINQ (Language-Integrated Query): Microsoft Developer Network [HTML] https://msdn.microsoft.com/ru-ru/library/bb397926.aspx

[3]. В.П. Иванников А.А. Белеванцев А.Е. Бородин В.Н. Игнатьев Д.М. Журихин А.И. Аветисян М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН, том. 26 (выпуск 1), 2014 г., стр. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7

[4]. Веб-сайт Coverity: Software Testing and Static Analysis Tools [HTML] http://www.coverity.com/

[5]. Веб-сайт Klocwork: Source Code Analysis Tools for Security & Reliability [HTML] http://www.klocwork.com/

[6]. Веб-сайт SonarLint [HTML] http://www.sonarlint.org/

[7]. Веб-сайт PVS-Studio for C/C++/C# [HTML] http://www.viva64.com/

[8]. Веб-сайт SharpChecker [HTML] http://sharpchecker.ispras.ru/ru/

[9]. Веб-сайт Visual Studio - Microsoft Developer Tools [HTML] https://www.visualstudio.com/ru-ru/visual-studio-homepage-vs.aspx

[10]. Веб-сайт Roslyn .NET Compiler Platform [HTML] https://github.com/dotnet/roslyn

[11]. Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д. Компиляторы. Принципы, технологии и инструментарий: пер. с англ. / под ред. И.В. Красикова. М.: ООО «И.Д. Вильямс», 2008. 1184 с.

[12]. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. 2000. In Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00). ACM, New York, NY, USA, pp. 264-280. doi: 10.1145/353171.353189

[13]. Falke Stephan, Merz Florian, Sinz Carsten. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. 2010. Tools and Algorithms for the Construction and Analysis of Systems, pp. 623–626. doi: 10.1007/978-3-642-36742-7_48

[14]. В.К. Кошелев И.А. Дудина В.Н. Игнатьев А.И. Борзилов Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды Института системного программирования РАН, том. 27 (выпуск 5), 2015.г., стр.. 59–86. DOI: 10.15514/ISPRAS-2015-27(5)-5

[15]. H. R. Andersen, An Introduction to Binary Decision Diagrams, 1997, Lecture notes [PDF] http://www.cs.utexas.edu/~isil/cs389L/bdd.pdf

[16]. K. L. McMillan, Interpolants from Z3 proofs, 2011. Formal Methods in Computer-Aided Design (FMCAD), pp. 19–27.

[17]. Описание стандарта ECMA-335 [PDF] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf

[18]. Веб-сайт библиотеки Mono.Cecil http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/

[19]. Описание Windows API в Microsoft Developer Network [HTML] https://msdn.microsoft.com/en-us/library/cc433218

[20]. Веб-сайт Библиотека классов .NET Framework в Microsoft Developer Network [HTML] https://msdn.microsoft.com/ru-ru/library/mt472912(v=vs.110).aspx

C# static analysis framework

V.K. Koshelev <vedun@ispras.ru>
V.N. Ignatyev <valery.ignatyev@ispras.ru>
A.I. Borzilov <helendile@ispras.ru>
ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. The paper describes static analysis techniques that are used for defect detection in C# programs. The goal of proposed analysis approaches is to catch more defects within an acceptable amount of time. Although the paper contains a description of full analysis cycle, it mainly focuses on special aspects that distinguish C# analysis approaches from well-known Java and C++ techniques. The paper illustrates methods that allow taking into account C# specialties of all analysis stages: call graph and control flow graph construction, data flow analysis, context- and path-sensitive interprocedural analysis. We propose an adaptation of symbolic execution methods inspired by Bounded Model Checking and Saturn Software Analysis Project. The paper also explains the organization of memory model, which is suitable for both a precise intraprocedural analysis and a creation of compact function-bound conditions used for interprocedural analysis. Special attention is paid to optimization of condition size and simplicity during a path sensitive-analysis. The conditions produced by a path-sensitive analysis are supposed to be solved by modern SMT solvers like Microsoft Z3 Prover. Different approaches to external functions modeling are covered. All proposed approaches are implemented in the SharpChecker static analysis tool and, as evaluated on several open source C# projects of varying size (1K - 1.35M lines of code), display good results and scalability.

Keywords: static analysis; null pointer dereference; path-sensitive analysis; context-sensitive analysis; function summary; bug detection; Roslyn; C#

DOI: 10.15514/ISPRAS-2016-28(1)-2

For citation: 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

References

[1]. TIOBE Index [HTML] http://www.tiobe.com/tiobe_index
[2]. LINQ (Language-Integrated Query): Microsoft Developer Network [HTML] https://msdn.microsoft.com/ru-ru/library/bb397926.aspx
[3]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v ishodnom kode programm (Static analyzer Svace for finding of defects in program source code). Trudy ISP RAN [Proceedings of ISP RAS], volume. 26 (issue 1). 2014. pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7
[4]. Coverity: Software Testing and Static Analysis Tools [HTML] http://www.coverity.com/
[5]. Klocwork: Source Code Analysis Tools for Security & Reliability [HTML] http://www.klocwork.com/

[6]. SonarLint [HTML] http://www.sonarlint.org/
[7]. PVS-Studio for C/C++/C# [HTML] http://www.viva64.com/
[8]. SharpChecker [HTML] http://sharpchecker.ispras.ru/ru/
[9]. Visual Studio - Microsoft Developer Tools [HTML] https://www.visualstudio.com/ru-ru/visual-studio-homepage-vs.aspx
[10]. Roslyn .NET Compiler Platform [HTML] https://github.com/dotnet/roslyn
[11]. Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2006
[12]. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. 2000. In Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00). ACM, New York, NY, USA, pp. 264-280. doi: 10.1145/353171.353189
[13]. Falke Stephan, Merz Florian, Sinz Carsten. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. 2010. Tools and Algorithms for the Construction and Analysis of Systems, pp. 623–626. doi: 10.1007/978-3-642-36742-7_48
[14]. V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov. Chuvstvitel'nyj k putjam poisk defektov v programmah na jazyke C# na primere razymenovanija nulevogo ukazatelja (Path-sensitive bug detection analysis of C# program illustrated by null pointer dereference). Trudy ISP RAN [Proceedings of ISP RAS], volume 27 (issue 5), 2015. pp. 59–86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-5
[15]. H. R. Andersen, An Introduction to Binary Decision Diagrams, 1997, Lecture notes [PDF] http://www.cs.utexas.edu/~isil/cs389L/bdd.pdf
[16]. K. L. McMillan, Interpolants from Z3 proofs, 2011. Formal Methods in Computer-Aided Design (FMCAD), pp. 19–27.
[17]. ECMA-335 Standard [PDF] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf
[18]. Mono.Cecil http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/
[19]. Windows API: Microsoft Developer Network [HTML] https://msdn.microsoft.com/en-us/library/cc433218
[20]. .NET Framework Class Library: Microsoft Developer Network [HTML] https://msdn.microsoft.com/en-us/library/mt472912(v=vs.110).aspx