FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Robert Husák

Source Code Assertions Verification Using Backward Symbolic Execution

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	RNDr. Jan Kofroň, Ph.D.
Study programme:	Software and Data Engineering
Study branch:	Software Engineering

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Source Code Assertions Verification Using Backward Symbolic Execution

Author: Robert Husák

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: In order to prevent, detect and fix errors in software, various tools for programmers are available, while some of them are able to reason about the behaviour of the program. In the case of C# programming language, the main representatives are Microsoft FxCop, Code Contracts and Pex. Those tools can, indeed, help to build a highly reliable software. However, when a company wants to include them in the software development process, there is a significant overhead involved. Therefore, we created a "light-weight" assertion verification tool called AskTheCode that can help the user to focus on a particular problem at a time that needs to be solved. Because of its goal-driven approach, we decided to implement it using backward symbolic execution. Although it can currently handle only basic C# statements and data types, the evaluation against the existing tools shows that it has the potential to eventually provide significant added value to the user once developed further.

Keywords: code analysis verification backward symbolic execution

Contents

Introduction

Taking part in the software development process is a hard task for everyone, including programmers. The rate of delays and errors they produce has a direct influence on the success of the project. Therefore, it is beneficial to provide them tools that can help them to reduce these factors.

Some of the tools aim to provide intuitive inspection and refactoring of the code. In the case of C# programming language, we can mention widely used *Resharper* and *CodeRush*. Such tools immediately provide the added value, because they extend the functionality of the integrated development environment (IDE) without any complicated setup.

There are also tools capable of reasoning about the behaviour of the application, for instance, *Coverity* and *Code Contracts static checker*. Coverity provides a list of potentially problematic places in the code. The contracts checker is sometimes able to automatically prove the contracts, helping the user to understand the logic of the code.

Those tools can, indeed, help to build a highly reliable software. However, when a company wants to include them in the software development process, there is a significant overhead involved. The effort of the programmers spent on specifying all the contracts or inspecting all the potential problems is too high.

Therefore, the goal of this thesis is to implement a "light-weight" behaviour verification tool. Instead of trying to find all the possible errors in the program, it will focus on one certain problem the user currently needs to solve.

In order to choose the right approach, we will analyse the current techniques used in program analysis. Also, we will inspect the tools currently available for C#, try to get inspired by them but avoid duplicating their functionality.

Organization

The thesis consists of three parts: Background, Solution and Evaluation. In Background, we summarize existing techniques and tools in the area of program analysis. Chapter 1 explains the satisfiability modulo theories problem and the semantics of its solvers. These solvers are used by some of the techniques described in Chapter 2. Chapter 3 presents existing tools and the way how they utilize the mentioned techniques to provide useful information to users.

The tool developed as a part of the thesis is described in Solution. Chapter 4 summarizes the requirements placed on it, suggesting what added value it can bring against the existing tools. Chapter 5 explains the approach chosen to reflect the requirements and provides a top level overview of the architecture of the tool. The implementation details in terms of particular modules and classes are reviewed in Chapter 6.

The purpose of Evaluation is to objectively assess whether the created tool fulfils its purpose and compare it with already existing tools. Chapter 7 outlines the scenarios for the experiment, whereas Chapter 8 presents its results. In Chapter 9, we interpret the results and suggest possible ideas for the future of the tool.

The last chapter concludes the whole thesis. The user documentation and content of the enclosed CD are described in the Attachments section.

Part I

Background

1. Satisfiability Modulo Theories

The reason we start by explaining this decision problem is to provide the base for some of the algorithms mentioned later, because they rely on satisfiability modulo theories (SMT) solvers. [1] We will describe its basic principles, standard SMT-LIB and an SMT solver from Microsoft called Z3.

1.1 Principles

An instance of the problem is a formula in first-order logic. Functions, predicates and sorts of the variables originate from the set of theories supported by the particular solver. It is also possible to define custom functions, predicates and even sorts by combining the existing ones.

The supported theories are mostly targeted at the verification of programs. Therefore, apart from linear and nonlinear arithmetic, there are theories for working with bit-vectors, sequences, records and other data structures. For example, array theory over an index type I and a value type V comprises a corresponding domain of arrays A, functions *read*, *write* and the following axioms: [2]

$$\forall a \in A, i, j \in I.(i = j \Rightarrow read(write(a, i, v), j) = v)$$
$$\forall a \in A, i, j \in I.(i \neq j \Rightarrow read(write(a, i, v), j) = read(a, j))$$
$$\forall a, b \in A.(a \neq b \Rightarrow \exists i \in I.read(a, i) \neq read(b, i))$$

There are three possible results the solver returns: satisfiable (SAT), unsatisfiable (UNSAT) and unknown. In the first case, the solver can also provide the model, e.g. the interpretation of the variables from the formula in their respective sorts. In the second case, because the formula is supplied to the solver as a conjunction of clauses, it can return their subset to mark which clauses caused the unsatisfiability. This subset is called unsatisfiable core. The last result is returned when the solver is not capable of finding the answer. The reasons may comprise the usage of undecidable theories, memory exhaustion or timeout.

1.2 SMT-LIB

In order to unite the syntax and semantics of various SMT solvers, SMT-LIB standard [3] was introduced. Its current version in the time of the writing is 2.5. The language it defines has a Lisp-like syntax in order to enable easy parsing. In Listing 1.1, we utilize it to define 2D integer vector sort together with some operations and prove three properties of the dot product: commutativity, distributivity over vector addition and the relation to scalar multiplication.

Listing 1.1: Example SMT-LIB code to prove three properties of the 2D integer vector dot product

```
(declare-datatypes () ((Vec (vec (x Int) (y Int)))))

(define-fun dot ((u Vec) (v Vec)) Int
  (+ (* (x u) (x v)) (* (y u) (y v)))
)
```

```
(define-fun add ((u Vec) (v Vec)) Vec
  (vec (+ (x u) (x v)) (+ (y u) (y v)))
)

(define-fun mul ((s Int) (u Vec)) Vec
  (vec (* s (x u)) (* s (y u)))
)

(declare-const c1 Int)
(declare-const c2 Int)
(declare-const a Vec)
(declare-const b Vec)
(declare-const c Vec)

(push)
(assert (not (= (dot a b) (dot b a))))
(check-sat)
(pop)

(push)
(assert (not (= (dot a (add b c))
                (+ (dot a b) (dot a c)))))
(check-sat)
(pop)

(push)
(assert (not (= (dot (mul c1 a) (mul c2 b))
                (* (* c1 c2) (dot a b)))))
(check-sat)
(pop)
```

At first, commands `declare-datatypes`, `define-fun` and `declare-const` are used in the beginning to provide us a way to express the properties. In order to share these definitions and declarations among all the verified properties, we utilize an assertion stack, which is realized by the `push` and `pop` commands. The former creates a "restore point", whereas the latter tells the solver to forget everything between it and the corresponding `push`. Then, using the `assert` command, we add the negation of each property to the current formula and check its satisfiability by `check-sat`.

As we can see, we have in fact performed a proof by contradiction. All the checks return UNSAT, which means that the negations are false, hence the properties are verified.

One of the solvers adhering to the standard is Z3 [4], a state-of-the-art SMT solver from Microsoft Research. It supports a wide range of theories, including bit-vectors, arrays, sequences or uninterpreted functions. Even custom theories are enabled, as the user can provide them as plug-ins. [5] It is published as an open source project with MIT licence. Various verification and test generation tools use Z3 as their backbone, for example SAGE, SLAM, Boogie or Pex.

2. Program Analysis

Program analysis discipline comprise various methods that reason about properties of programs. In particular, we are interested in those that analyse program correctness. Most of them runs in the compile-time, without the need to launch the actual program. This chapter reviews the most important techniques used in practice: model checking, symbolic execution, abstract interpretation and data flow analysis.

2.1 Model Checking

This approach translates the program to a directed graph of states. Every state contains the current instruction pointer and the values of all the variables present in the program. Having the graph, we can validate specific properties in it using propositional logic. For example, we can check that a certain boolean variable is never *true* when another one is $false$. To express how the properties influence each other over time, there are various temporal logics. [6]

The main problem of model checking is apparently the construction of the model, because it grows rapidly with the size of the program and it can even be infinite. This phenomenon is often referred to as the state explosion problem. There are various approaches to alleviate it, one of them is the usage of symbolic states. Instead of particular values, they hold symbolic constraints, so that every such state can represent a large set of the conventional states. Another technique is the simplification of the model using an abstraction, which usually involves merging states in a way that some of the original properties are lost.

Despite such improvements, model checking is practically usable only on small programs with bounded state space. Good example are embedded systems or hardware drivers.

2.2 Symbolic Execution

Whereas model checking puts emphasis on the particular states of the program and transitions between them, the key concept of symbolic execution is the inspection of the possible execution paths. Symbolic execution can be either forward or backward. [7, 8]

Forward symbolic execution mimics a normal execution of the program, supplying the input in the form of a set of named variables instead of the particular values. Supposing that the program is completely deterministic, every value along the execution path can be expressed as a formula created from the input variables and appropriate operations. Whenever the algorithm encounters a branching on a condition, the current execution path splits into two: in the first one the condition must be true, whereas in the second one it must be false. The conjunct of all such conditions on the path is called a path condition. In order to check the reachability of a certain path, its path condition can be examined by an SMT solver. The possible corresponding model then contains the input values needed to steer the computation along the given path. As we can see, forward symbolic execution can be used to obtain a general picture of how the program works, possibly discovering unreachable code or reachable error situations.

Backward symbolic execution operates on execution paths as well, but it constructs them differently. Instead of starting at the entry point of the program, it begins the inspection at a given statement. The execution paths are then constructed backwards,

inspecting the possible ways how it might be reached. If and only if the statement is reachable, there exists at least one execution path from the entry point to the statement with a satisfiable path condition. Backward symbolic execution is apparently more goal-driven than the former, focusing only on one target at a time.

Although the exploration of the space state using symbolic execution may be more efficient than in the case of model checking, the principle of the state explosion problem still last. In this context it is called the path explosion problem, because it relates to the rapidly increasing number of possible paths in more complicated programs. There are various attempts to mitigate it while preserving the soundness of the algorithm, for example state merging: When two paths meet at a certain statement, it might be beneficial to merge them into one and continue exploring only with it instead of duplicating the similar exploration. This approach might lower the number of calls to SMT solver; however, every such call will be more complicated. Therefore, to decide whether to merge or not, an appropriate heuristic should be provided. [9]

Nevertheless, if the program contains data dependent loops or recursion, we usually need to sacrifice the soundness anyway in order to finish in a finite and reasonable time. There are also situations where we cannot model all the operations precisely enough. For example, the program may interact with the file system or even a service running on another computer.

For such cases, it is often useful to introduce a real execution phase into the algorithm, which launches the program with certain inputs, and to trace its execution. SMT solver can then be used to generate new inputs that will steer the execution path differently. This technique, called dynamic symbolic execution or concolic testing, is used to automatically generate test inputs maximizing the resulting code coverage. Although it works only for forward symbolic execution, a slightly different version for backward symbolic execution was introduced in [10]. We can then use such technique to address only the uncovered code.

2.3 Abstract Interpretation

As seen from the previous approaches, it is very difficult, resource-intensive and often even impossible to capture the exact semantics of a program, especially when it contains loops or recursion. On the other hand, if we omit a certain level of precision, some features of the program can be verified relatively easily and effectively, such as whether is every variable initialized before its first usage. These algorithms often share the same structure and differ only by the domain they are operating on. Abstract interpretation is a framework that summarizes this algorithm and specifies the requirements that must be fulfilled so that the algorithm is sound and finite. [11]

Before we can start the algorithm itself, a control flow graph (CFG) of the analysed function must be created. Each node in the CFG represents a statement, edges stand for the conditional and unconditional jumps between these statements. There can be also special nodes to mark the entry point of the function, return statements, function calls, exceptions etc. If we want to perform the analysis in the interprocedural way, we can interconnect the appropriate call, entry and return nodes of the particular graphs.

Having the CFG, we associate each edge with an initial empty value from the domain given by the particular algorithm. In the case mentioned earlier, called definite assignment analysis, the domain can be the power set of the variables defined in the function. When ordered by inclusion, this domain forms a complete lattice, which is very important, as we will see.

Next, a system of equations is created that reflects the semantics of the statements. For the purpose of our example, suppose we have a node n containing an assignment of a variable v, its ingoing edges i_1, i_2 and an outgoing edge o. If f denotes a function assigning a value from the domain to an edge, the equation corresponding to o will be $f(o) = \{v\} \cup (f(i_1) \cap f(i_2))$. The $\cap$ operator used for joining the path from two locations is particularly important, because it is monotone with respect to the order defined earlier. Therefore, when we run an iterative computation afterwards, abstract interpretation guarantees that after a finite amount of steps it will reach a fixpoint - a state not changeable by reapplying any of the equations.

While the previous example operated on a small finite domain, there are various abstractions taking place in larger and even infinite domains. Let us consider the modelling of a variable value using intervals. While the operations performed on the variable can be simulated using interval arithmetic, a problem arises when we start using loops. For example, in the case of an infinite loop incrementing the variable in its body, the algorithm in its current form would not end. Therefore, abstract interpretation provides an operation named widening that provides an over-approximation of the values. When inserted into appropriate places in the system of equations, it sufficient to prevent the endless loop. To alleviate the impact of the widening on the precision of the result, another operation called narrowing is introduced. It can be applied after the widening, performing another round of fixpoint computation.

There are various abstract domains addressing various aspects of the program. To abstract values of numeric variables, the already mentioned domain of intervals can be used. However, its main problem is that the relations between the particular variables are not preserved. Therefore, relational domains such as linear equalities, congruence relations or convex polyhedra were introduced.

Alias analysis inspects the reference variables in the code, assessing whether they can refer to the same object. Although its flow-insensitive variant does not necessarily use abstract interpretation, the flow-sensitive one commonly does. [12, 13, 14] As the number of objects allocated within a program is unbounded in general, the domain is infinite in this case. Therefore, an appropriate widening must be used.

Abstract interpretation can also be used as a base of general reasoning about the semantics of the program. Such algorithms attempt to capture it using Floyd-Hoare logic [15] and then prove various properties such as termination or correctness. They expect the programmer to provide sufficient hints in the form of assertions, assumptions, preconditions, postconditions, object invariants and loop invariants. An assertion is a boolean expression stated to be true in the given position in the program. A precondition is a requirement specified by a function to hold before it is called. A postcondition, on the other hand, is a guarantee the function provides after it ends. An assumption is a statement expected to be true without actually proving it. The purpose of an object invariant is to specify the properties the object must hold between the calls of its public methods. A loop invariant is a condition that must hold before and after each iteration of the loop. Apparently, proper loop invariants are crucial to prove the termination of the program.

The problem of this approach is that it requires a significant effort in providing these hints to make it work properly. Therefore, there are various attempts to infer some of them automatically, usually utilizing other abstract interpretation techniques. [16]

2.4 Data Flow Analysis

This category of analyses encompasses diverse techniques practically used in compilers to prevent certain well-defined errors or enable code optimizations. Conceptually, all the algorithms mentioned below adhere to the principles of abstract interpretation, usually using a finite domain and always operating on the intraprocedural level. We decided to emphasize them as a separate group, because of their simplicity and efficiency. The most common data flow analyses are reaching definitions, liveness analysis, available expressions, constant propagation and already mentioned definite assignment analysis. [17]

When a new value is assigned to a variable, its previous value is no longer reachable by the following statements. The purpose of reaching definitions is to discover which definitions of the variables are available for each statement. The results can be used for example to move a loop-invariant statement before the loop. This analysis is related to a static single assignment (SSA) form, where every variable is split into several versions.

On a certain code location, live variables hold values that may be used in the future. Liveness analysis results into a list of live variables for every location. In contrast to the already mentioned analyses, it is performed backwards. The result can be used to skip unnecessary writes to the memory or to enable two variables to share the same space.

Available expressions provides the compiler with the information about the expressions that do not need to be recomputed in each point of the program. The results of such expressions can then be reused.

When the value of a certain variable can be computed during the compilation, such process is called constant folding. Constant propagation was created for situations when there is a whole dependency chain of such variables. The result of this algorithm is the map of the variables to the corresponding constants. The compiler can use it to replace the reading of such values by directly using the constants. Furthermore, in case of boolean constants of branch conditions, it can eliminate unnecessary jumps, discovering unreachable code.

As mentioned in the previous section, definite assignment analysis is performed to ensure that every variable is initialized before it is used. When combined with constant propagation, it can provide even more useful information. [18]

3. Related Work

As the tool we want to develop is aimed at C#, we are mainly interested in the current program analysis technologies that can be used for this language. Those technologies, all created by Microsoft, comprise FxCop, Roslyn, Code Contracts and Pex. In the last section, we also mention some of the important technologies used for other languages and frameworks.

3.1 FxCop

FxCop [19] is a static program analysis tool created by Microsoft, incorporated into Visual Studio since version 2005 as a feature called Code Analysis. The programmer can select a set of rules to be verified upon every compilation.

FxCop is then added as a build step following right after the compilation. The reason for such approach is that it performs the analysis on the Common Intermediate Language (CIL) resulting from the compilation. To map the locations back to the source code, program debug database (PDB) files are used.

Particular rules vary be the difficulty of the corresponding analysis needed to verify them. As an example, while the analysis of the compliance with naming standards inspects only one identifier name at a time, some rules need to reconstruct the expressions and statements from CIL. Furthermore, advanced rules also utilize data flow analysis, e. g. to check that all the reference arguments of public methods were validated not to be null.

More than 200 rules are available, categorized in areas such as interoperability, design, globalization, naming, performance, security etc. A company can select particular rules to enforce in the code and also create custom rules built upon the same engine.

The problem of FxCop today is that C# features such as LINQ[1] or asynchronous methods significantly complicate the resulting CIL and it is difficult to infer its original semantics. That is one of the main reasons why its development no longer continues. [20]

3.2 .NET Compiler Platform ("Roslyn")

In order to provide programmers with the analysis of their C# code while they work in an IDE, there were originally two approaches. The first one, as seen in the case of FxCop, was to run after the compilation and analyse the resulting CIL. The second one was to run on background in the IDE and continuously analyse the semantics of the code in a similar way to the compiler. The advantage of the second approach is that it allows more interactivity and simplifies the provision of automatic fixes to the code. The obvious disadvantage is the duplicity of source code processing. Therefore, Microsoft enabled also a third approach by creating .NET Compiler Platform, commonly known as Roslyn. It is an open source compiler of C# and Visual Basic running directly in the IDE, providing various services and extension points. As a result, the analyses can reuse the information gathered by Roslyn, removing the duplicity of the code processing. [21]

To explain the services provided by Roslyn, let us analyse the steps it performs during the compilation process. As every compiler, it starts by parsing all the source files in the project and creating an abstract syntax tree (AST) for each one of them. All the possible

[1] Language Integrated Queries

syntax errors are reported during this stage. In the next phase, called binding, Roslyn assigns particular symbols and operations to the statements producing a semantic model. Various errors and warnings can arise there, such as type mismatches, unnecessary casts etc. Furthermore, the CFG of each method can be then examined by data flow analysis, particularly by definite assignment analysis and constant propagation. The last step is to emit the semantic model to the corresponding CIL. During the time when the programmer is writing the code, the CIL emission is not necessary. Instead, only the preceding phases can run in cycles, providing code diagnostics.

The most natural way to extend Roslyn using its API is to implement custom code analysers that provide code diagnostics to the programmer. They can be hooked to be launched on certain AST nodes, possibly using the information from the semantic model. The CFG is not directly available through the API; however, some limited information about the data and control flow can be obtained. Another popular extension points are custom refactorings and code fixes. Both can be used to update the code on demand, while the latter relate to the code analysers, because they relate to certain warnings or errors produced by them. For more complicated usages of Roslyn, a Microsoft Visual Studio extension that accesses it directly can be created.

In contrast to FxCop, Roslyn is still being actively developed by Microsoft and its usage as a platform for code analysis is heavily propagated and endorsed. At the time of writing, a significant number of FxCop rules has already been converted to Roslyn analysers and more are planned. [22] Another example of a popular code analysis tool converted to Roslyn is StyleCop. As its name suggests, its rules are aimed at the visual aspect of the code. Apart from these general-purpose analysers, some libraries also provide analysers to ease their usage, for example Entity Framework or Roslyn itself.

3.3 Code Contracts

The aim of Code Contracts is to enable programmers to annotate their code to express certain requirements or semantics in .NET. In particular, these annotations take the form of assumptions, assertions, preconditions, postconditions and object invariants. For example, the following C# code at the beginning of a method can be used to express a precondition: `Contract.Requires(x >= 0);` Code Contract provide three tools to utilize the annotations: a documentation generator, a runtime check generator and a static checker. [23]

The last one mentioned performs the most interesting and complicated task. Its purpose is to discharge proof obligations emerging both from the contracts themselves and from various possible errors specified by the user, for example null dereferencing or array bounds exceeding. Let us focus on how the static checker operates. First, after reading the analysed assemblies it attempts to sort the methods so that each method is processed before it is called from another method. When processing a method, its CIL is loaded into the memory and the CFG enriched by the provided contracts is created. In particular, each contracts is converted either to an assumption, or to an assertion. For example, to the call site of a method are its preconditions inserted as assertions and its postconditions as assumptions. Having this set up, an abstract analysis over various domains takes place inferring facts about the code. These facts comprise loop invariants, possible ranges of numerical values, aliasing information etc. Next, the proof obligations are collected and attempted to be discharged by the inferred facts. Those that cannot be discharged even by a refined analysis are reported to the user. Ultimately, the inferred facts are used to derive additional postconditions of the method and the next method in the sequence is processed. [24]

As a result of the analysis, we can see which proof obligations were not discharged, some of them may be even proven to be false. Furthermore, various hints of what preconditions, postconditions and object invariants to add are displayed.

3.4 Microsoft Pex

Microsoft Pex [25] is a tool for automated test input generation, currently known as the IntelliTest feature in Visual Studio 2015 Enterprise. [26] It is based on aforementioned dynamic symbolic execution.

At the beginning of the process, a set of initial values is passed to the examined method. The execution is then observed using a library called Extended Reflection. Using the .NET profiling API, it instruments the code to send information to a symbolic interpreter running on background. As a result, the interpreter is aware of all the branching conditions and can use Z3 to suggest new inputs to steer the execution towards the unexplored branches.

This way, Pex is capable of examining paths that would be unable to model by symbolic execution alone. It only requires the externally called methods to be deterministic. It contains a mechanism to detect a violation of that rule and warn the user.

3.5 Other

There are many other inspiring tools not related to Microsoft or .NET, such as Coverity, Symbolic PathFinder or KLEE. All of them are used by companies such as NASA[2], implying their reliability is significant. [27, 28, 29] Coverity is a commercial static code analysis tool used to find various defects, for example null pointer dereference, resource leak or buffer overrun. [30] Symbolic PathFinder [28] exercise Java bytecode programs by symbolic execution and model checking to automatically generate test cases and detect errors. It can handle complex mathematical constraints, aliasing, multithreading etc. KLEE [31] is a symbolic virtual machine capable of modelling the whole environment of complex system programs, e. g. the networking and filesystem. [32] As a result, it can generate high-coverage tests, find bugs and verify consistency of different programs with the same interface.

[2]National Aeronautics and Space Administration

Part II

Solution

4. Requirements

Having reviewed the current state-of-the-art approaches in the previous part, we will explain our solution and its inner workings. While the following chapters describe the design and implementation of the solution, this one summarizes the requirements we have specified beforehand.

The first section contains a general description of the features we want the solution to contain. Next section reminds some of the most important non-functional requirements, sometimes called qualitative attributes. [33] The last section suggests some ideas for the future development which the design should be open to.

4.1 Functional Requirements

The primary goal of the tool is to simplify programmers' work by helping them to fix the current errors and preventing them from introducing new ones. As we have seen, there is already a plenty of tools created for this purpose, so we want to delimit against them, trying to complement their functionality and extract the most useful features for our case from each one.

The first point to emphasize is the *minimization of prerequisites*. In order to find errors using Pex, we need to create unit tests and adapt them to be usable by it. To practically use the Code Contracts static verification, we have to create the contracts for the procedures involved. We want to prevent the need of such time-consuming tasks. Our tool should be able to perform a solid work with a minimal setup, with a possibility to adjust it later to achieve better results.

A lot of tools operate on the program as a whole, trying to report all the potential errors. This approach is useful for companies that have enough resources to inspect all these reports and solve the possible problems as a part of their quality assurance procedures. However, to find a cause of a particular error, it is more useful to use a *demand driven* approach, which is the way we want to take. Its neat effect is also that it can reduce the complexity of the analysis, because some of the program parts need not to be analysed at all.

On the other side, in order to be useful, our tool must work in a computationaly expensive *interprocedural* way, e.g. taking into account the execution flow through multiple procedures.

As the reasoning about a program execution according to its source code is generally undecidable, the tool might often run into a situation where it cannot provide an exact answer. Even in this case it must *provide detailed information* about the reason of the failure to the user. The same counts also for the case of successful analysis, where it must provide a detailed result.

4.2 Non-Functional Requirements

This type of requirements is not related to the features the application provide. Instead, they refer to its general qualities by specifying criteria it needs to pass. In our case, we will not specify those criteria in exact terms, but rather remind some of such requirements so that we keep them in mind when creating the tool.

The first such requirement is *usability*. It is crucial that the user is capable of using the tool intuitively and that the tool provides them relevant information. The learning

curve of the tool usage must not be too steep. The basic features have to be reachable by a wide range of programmers with varying experience.

A closely related requirement is *stability*, especially if the tool is integrated into an IDE. If an unhandled exception or other problematic scenario occurs, it may crash the entire IDE. The user will not only be disturbed, but they can also lose the unsaved work. Therefore, it is important to locate the errors as early as possible and try to recover from them. If the recovery fails, it is still better to shut down or restart just the tool than the entire IDE.

The usability of the tool is also closely related to the ability to perform the analysis quickly enough, e.g. to *performance*. In our case it depends mostly on the selected algorithms and technologies, the computing power of the user's computer and the complexity of the particular task, so it might be hard to provide any guarantees about it. However, we can alleviate its impact on usability if we keep the user informed about the current state of the analysis.

In order to prevent errors in the code and user experience, the practice shows it pays to use several kinds of tests, such as unit tests, integration tests, GUI tests etc. A requirement related to this is called *testability*. Its point are not the tests itself, but the characteristic of the code as being "testable". Therefore, the classes should be loosely coupled, reasonably using interfaces, easily mockable and capable of providing the information of their inner state to the testing framework.

Among others, loose coupling and proper interface usage are also related to *extensibility*, i.e. the ability to easily change functionality and add new. That is crucial for gradually improving the tool according to the needs of the users. Some of the ideas for future extensions are provided in the next section, ordered by priority.

4.3 Future Development

Because of the limited scope of the thesis, modelling *references and custom data types* is not expected to be supported at this time. However, those features are fundamental for the practical usage of the tool; therefore, it will probably be its first extension.

We have already mentioned that the user should be informed about any problems during the analysis, such as infinite loops or operations too complicated to model. An idea to enhance the user experience in such cases is to enable them to influence the search in an *interactive* way. For example, the user may skip some execution paths or provide assumptions that might ease the analysis.

Such feature would affect a single analysis, but its direct effect will not be permanent. For that we may introduce *code changes suggestions*, such as adding input checks, assertions, loop invariants or condition simplification. The tool would gather knowledge for these according to the structure of the program and the error being inspected.

Another planned feature is the support of *multiple programming languages*. Although it might not be in the nearest future, it is important not to block this feature by coupling the tool with a particular programming language or framework.

5. Architecture

This chapter presents the idea of assertions verification using backward symbolic execution, integrated as an extension of an IDE. The first section explains the idea and validates it against the requirements specified in the previous chapter. In the remaining sections, we present the core algorithms of the solution on a top level approach, e.g. not considering the particular modules and classes. Such implementation details will be discussed in the following chapters.

5.1 Approach

From now on, we will refer to the tool as to AskTheCode, because it is the name we selected for it. It reflects its main mission: to provide a dialogue about the semantics of the program to the user. It also purposefully sounds like that the program was responding on its own, hence not needing any complicated setup.

Considering the current technologies and the requirements placed on AskTheCode, we decided to use backward symbolic execution as the fundamental algorithm. It starts on a specific assertion selected by the user, making its negation the initial path condition and subsequently traversing against the flow of the program until it reaches an *entry point*. This term usually denotes `main()` function in the case of an application and public functions in the case of a library. AskTheCode is integrated into an IDE, enabling it to be used in the development process seamlessly.

To justify the approach, we will validate it against the functional requirements. There are no significant prerequisites, the only one being writing down the assertion to verify, if it is not present yet. In fact, we think of this as of a useful feature, because the presence of the assertion might later give the programmers updating or reviewing the program a hint about the inner workings. The solution is demand driven thanks to the nature of backward symbolic execution, enabling it also to be performed in an interprocedural way.

At the current level of abstraction, we cannot directly validate the non-functional requirements, as they are more related to the detailed implementation. As such, they will be referred in the following chapters to justify some of the design decisions.

Speaking about the design on a top level, we decided to divide the task into three distinct parts. Because the symbolic execution cannot be performed directly on the program being analysed, the first part takes care of transforming it into a set of symbolic CFGs representing the particular functions. The second part performs the symbolic execution itself, returning the results and/or errors gathered along the way. The last part is responsible for interacting with the user, gathering the input and presenting the output. Those parts are further explained in the following sections.

5.2 Control Flow Graph Generation

The main purpose of this part is to transform the given program to a collection of symbolic CFGs that can be traversed by the path exploration. Another task we demand from this part is to generate mappings of the symbolic CFGs back to the source code so that we can later properly display the results of the exploration. For compiled languages, there are basically two options of how to perform these tasks: we can either analyse the compiled program or its source code.

The first method might be easier, because the semantics of the machine code or the intermediate representation is supposed to be more straightforward to understand and often less complicated than the original source code. To map it back to the source code, one could use the debugging information provided by the compiler.

Listing 5.1: Example C# code to generate CFG from

```
public static int QuadraticFirstRoot(int a, int b, int c)
{
    int D = b * b - 4 * a * c;
    int dividend = 0;

    if (D > 0)
    {
        dividend = -b;
        dividend += (int)Math.Sqrt(D);
    }
    else if (D == 0)
    {
        dividend = -b;
    }
    else
    {
        Debug.Assert(D < 0);
        Trace.WriteLine(-1);
        throw new ArgumentException();
    }

    int divisor = (2 * a);
    Debug.Assert(divisor != 0);
    return dividend / divisor;
}
```

That method is popular among various code analysis tools, in our case, however, we decided to prefer the second one. This decision will be further justified in the next chapter, as it is related to particular technologies: Microsoft Visual Studio and Roslyn. The first method can still be implemented in the future, as it might be helpful in the case the user references a library without having access to its source code.

To properly describe the approach we have chosen, let us examine the source code shown in Listing 5.1 and the corresponding CFG in Figure 5.1. It is a simple static method written in C# that returns an integer approximation of the first real root of a quadratic equation. If the equation does not have any such root, an argument exception is thrown. The CFG of the method is assembled from the nodes of five types and the edges between them. The variables and expressions used throughout the CFG correspond to SMT-LIB standard, even though we do not display them in the Lisp-like syntax to improve readability.

Every CFG contains a single *enter node* determining the starting point of the computation. In our notation, it is marked by the corresponding "enter" keyword, the name of the method and the list of parameters. The *return node* is similar to the corresponding language feature. It must contain an expression of an appropriate type, if the method is not of the "void" type. The *call node* is strongly related to both of the mentioned, as it express calling a method with certain arguments and optionally storing its result. The exception handling is currently not fully implemented, because to do it properly, a complicated analysis of the code would be needed. Therefore, only the *throw node* is now present, which denotes an exception to be thrown and optionally any arguments to be passed to its constructor.

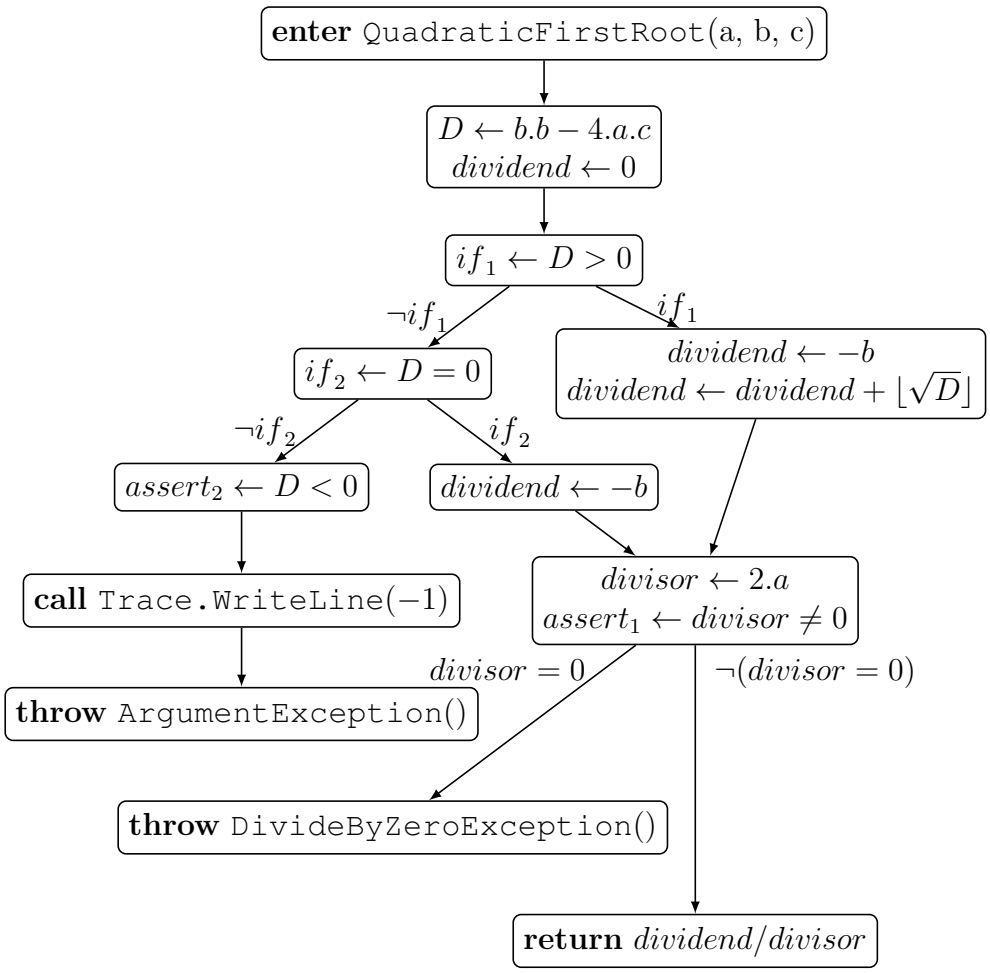

Figure 5.1: Example CFG generated from C# code in Listing 5.1

The most frequent type is the *inner node*, recognizable by not having any leading keyword. Instead of marking any interprocedural flow as the other types, it just contains a list of assignments. Each assignment contains a variable on the left side of the arrow and an expression of the same sort on the right side. The node does not take care of the variable declaration and does not distinguish between parameters, declared variables and temporal variables.

Directed edges between the nodes depict the control flow. Every edge can have a boolean expression attached to itself, in which case it is used only if the expression holds. If there is no condition, tautology is assumed. All node types except for return and throw must have one or more outgoing edges with mutually disjoint conditions and the disjunction of those conditions must be equal to tautology.

Having inspected our representation of CFG, we proceed to its construction. At this point we expect to use an external tool such as Microsoft Roslyn to help us analyse the syntax and semantics of the code. Otherwise, the complexity of AskTheCode would be similar to the one of a compiler, so it would have been better to analyse the result of the compilation instead, as mentioned before.

Supposing we have the required tool, we can use it to get the AST of the method and traverse it in a recursive way. The control flow statements and expression evaluations will gradually expand, eventually forming the basic shape of the graph. Once it is completed, we can generate the mapping back to the source code by examining the positions of the particular AST nodes corresponding to the CFG nodes.

To translate variables, constants and operations on them from the programming lan-

guage to the format we support, we use entities called *type models*. Each type model provides a mapping from the expressions in the code to their representation in SMT-LIB standard. Besides that, it can model the operations performed on that type. In the example, the integer type model is responsible for properly converting all the operations such as multiplication, addition, comparison or division. The division is especially interesting, because it is not defined for all the inputs, throwing an exception if the divisor is zero. [1] Therefore, the operation modelling must be able to provide the information about such behaviour, too. Another important feature of a type model is that when given the interpretations of its variables, it displays the value that would have the type it is modelling had.

As we can see in the case of comparisons or type conversion, the operations can produce models of different types. We can also model static methods, as seen in the case of `Math.Sqrt()` and `Debug.Assert()`. Assertions do not have any special meaning in the CFG, their results are just stored to helper variables so that we can select one of such assignments as a starting point of the path exploration, as described in the next section.

5.3 Path Exploration

This algorithm receives the system of CFGs and the assertion to verify as the input and it is expected to produce a collection of valid execution paths breaking it. Optionally, it can also explain why are some of the considered paths not feasible by highlighting the contradictory statements.

An example of breakable assertion from the CFG in Figure 5.1 is $divisor \neq 0$, stored to variable $assert_1$. There are two execution paths leading to its breaking from the enter node, both having a equal to zero. They only differ by b being zero in one case and non-zero in the other, which yields into taking different edges from the $if_1 \leftarrow D > 0$ node. The assertion $assert_2 \leftarrow D < 0$ can be proven as being unbreakable, because its negation together with the two nodes, closest against the control flow, forms a contradiction.

The fundamentals of path exploration are depicted in Algorithm 1, inspired by a generic symbolic exploration algorithm found in [9]. Although our algorithm is based on backward symbolic execution instead of the forward one, some of the principles are similar.

We begin by explaining the used terms and the meaning of the input. Terms *location* and *path* are strongly related. The first one refers to a CFG node with a slight simplification for the sake of the algorithm's conciseness: We assume that inner nodes can only contain one assignment. A node with multiple assignment is semantically equivalent to a sequence of such nodes.

Path is either a simple sequence of locations or an acyclic graph resulting from path merging. Only paths with the same last location can be merged, every path node can then be a successor of more than one path. Extracting the last location from a path is done by the *location* function.

In the algorithm, we manage a worklist of exploration states named w. Each state (p, pc, m) is represented by its path p, path condition pc and variable version mapping function m. Location l_0 represents the location to start the search from and expression a_0 is the assertion to verify.

[1] To be precise, CFG in Figure 5.1 should have contained also a check of valid value before converting the `double` return value of `Math.Sqrt` back to `int`. However, we decided to skip it for the sake of conciseness.

Algorithm 1 The path exploration core

```
1: // Set up the starting state
2: p_0 := (l_0)
3: w := {(p_0, ¬a_0, λv.0)}
4: while w ≠ ∅ do
5:     (p, pc, m) := pickNext(w)
6:     w := w \ {(p, pc, m)}
7:     // Process the successors of the selected state
8:     for all {e ∈ ingoingEdges(location(p)) : doFollow(e, p, pc)} do
9:         p' := extendPath(p, e.from)
10:        (pc', m') := addEdgeCondition(p, pc, m, e)
11:        // Process the assignment if present
12:        if content(e.from) is v ← e then
13:            m' := m[v ↦ m(v) + 1]
14:            pc' := addCondition(p', pc', v^{m(v)} = versionedExpression(e, m'))
15:        end if
16:        // Optionally merge the successor with another state in the worklist
17:        if ∃(p'', pc'', m'') ∈ w : doMerge((p', pc', m'), (p'', pc'', m'')) then
18:            w := w \ (p'', pc'', m'')
19:            (p', pc', m') := merge((p', pc', m'), (p'', pc'', m''))
20:        end if
21:        // Check the path condition if needed and optionally report the result
22:        if isEntryPoint(location(p')) then
23:            reportResult(p', pc', checkSat(pc'))
24:            continue
25:        else if doCheckSat(p', pc', m') then
26:            r := checkSat(pc')
27:            if r ≠ SAT then
28:                reportResult(p', pc', r)
29:                continue
30:            end if
31:        end if
32:        // Add the successor to the worklist
33:        w := w ∪ {(p', pc', m')}
34:    end for
35: end while
```

As we can see on line 3, the worklist starts only with the state containing the initial location and the negated assertion. Until the worklist is empty, we repetitively select a state using the *pickNext* function on line 5 and process it. This function is designed as one of the heuristics that can be injected to the algorithm. Another one, *doFollow*, can prevent the algorithm from exploring unreasonable paths. For example, it can bound the number of times a cycle is unwound.

It is used on line 8 to filter the results of the *ingoingEdges* function. For the most of the CFG node types, this function just returns its ingoing edges in the current graph. However, enter and call nodes are handled separately by creating custom temporary edges that model the interprocedural flow. The *addEdgeCondition* function on line 10 finishes this task together with optionally updating the variable version map and the path condition according to the edge.

On lines 12-15, an assignment of an expression to a variable is processed. For the case the variable appears in the expression, we need to make sure its version is different there by incrementing it in the version map. As a result, the path condition contains variables in SSA form with version numbers increasing against the control flow.

The resulting state can be then merged with another state on the same location. As we can see on lines 17-20, we can influence the merging by the heuristic function *doMerge*. Merging itself, provided among others by *merge* function, will be explained below.

On lines 22-31 is depicted the process of checking the satisfiability of the path condition and possibly reporting the result. If we reached the entry point, we always perform the check, report its result and continue without adding the state to the worklist. The same situation happens, when the heuristic function *doCheckSat* tells us to perform the check and it is either unsatisfiable or unknown. When implemented properly, it can help the algorithm to cut the infeasible path early on, saving resources.

The *reportResult* function analyses the result of the satisfiability check. In the case of SAT, it converts the model to an execution path with corresponding values assigned to each variable version along the way. If the result is UNSAT, the checked path and the nodes and edges corresponding to the unsatisfiable core are returned. If the result is unknown, only the path and a description of the reason is returned.

If the result was not reported, we add the successor to the worklist on line 33 and loop again.

A simple example of path exploration from $assert_1$ in Figure 5.1 is provided in Figure 5.2. We assume *doFollow* and *doCheckSat* always return *true*, while *doMerge* returns $false$. Internally, the path condition is checked by a set of incremental SMT solvers, more information about their life cycle will be provided in the next chapter.

On 5.2a and 5.2b we can see the assertion stacks of both paths. In the left column of an assertion stack, groups of assertions are gradually added line by line. The right column contains the result of the satisfiability check of the current assertions in the stack, i.e. the assertions on the same line and all the lines before. The conditions extracted from every CFG node and its leading edge are added to the stack as one level. The version of each variable is displayed as its upper index.

After both explorations reached the entry point, their models shown in 5.2c and 5.2d are generated. The found path and the corresponding model provide enough information to later show the detailed information about the execution to the user.

In Figure 5.3, the assertion $assert_2$ is examined. As we can see, the condition of the only path leading to it proves to be unsatisfiable; therefore, the assertion is verified.

Looking at both paths explored from $assert_1$ in Figure 5.1, we see that some of the nodes are shared between them, therefore analysed twice. The more complicated

$\neg(divisor^0 \neq 0)$	SAT
$divisor^0 \leftarrow 2.a^0$	SAT
$dividend^0 = -b$	SAT
if_2^0 $if_2^0 = (D^0 = 0)$	SAT
$\neg if_1^0$ $if_1^0 = (D^0 > 0)$	SAT
$dividend^1 = 0$ $D^0 = b^0.b^0 - 4.a^0.c^0$	SAT

(a) Assertion stack of the first path

$\neg(divisor^0 \neq 0)$	SAT
$divisor^0 \leftarrow 2.a^0$	SAT
$dividend^0 = dividend^1 + \lfloor\sqrt{D^0}\rfloor$ $dividend^1 = -b^0$	SAT
if_1^0 $if_1^0 = (D^0 > 0)$	SAT
$dividend^2 = 0$ $D^0 = b^0.b^0 - 4.a^0.c^0$	SAT

(b) Assertion stack of the second path

a^0	0
b^0	0
c^0	0
D^0	0

$dividend^1$	0
if_1^0	$false$
if_2^0	$true$
$dividend^0$	0
$divisor^0$	0

(c) Model of the first path

a^0	0
b^0	1
c^0	0
D^0	1

$dividend^2$	0
if_1^0	$true$
$dividend^1$	-1
$dividend^0$	0
$divisor^0$	0

(d) Model of the second path

Figure 5.2: Path exploration from $assert_1$ in Figure 5.1 without merging

the graph gets, the more redundancy might be potentially produced, contributing to path explosion. The first attempt to alleviate its impact is by using the incremental solver as economically as possible. For example, instead of starting the solver again after exploring the first path, we can just retract it to the last branching node by appropriately popping its assertion stack. The solver then might take advantage of the information it already gathered about the assertions left in the stack. In our case, it would not need to repeatedly solve the assertions in the first two lines of Figure 5.2b, possibly knowing they imply $a^0 = 0$.

Although that should help in some cases, we see the last two nodes in the explorations still need to be analysed twice and the path explosion was not alleviated at all. Therefore, the algorithm was created to support exploration state merging, mentioned for example in [9]. As it is beyond the scope of this thesis, the heuristics functions $doExpectMerging$, $doMerge$ currently always return false so that we never explore a merged path.

$\neg(D^0 < 0)$	SAT
$\neg if_2^0$ $if_2^0 = (D^0 = 0)$	SAT
$\neg if_1^0$ $if_1^0 = (D^0 > 0)$	UNSAT

Figure 5.3: Path exploration from $assert_2$ in Figure 5.1

5.4 User Interface

Having reviewed the fundamental algorithms, we will explain how are they used to provide the user experience. The approach is rather conceptual and programmer-oriented; to see the Graphical User Interface (GUI) from the user's perspective, see the User Documentation attachment.

Launching the exploration must be as straightforward as possible: After selecting the statement to start the exploration from, it can be done either by a special keyboard shortcut, from a context menu in the code editor or from a button on the main panel.

Supposing the selected statement is located inside a method, AskTheCode generates its CFG and then uses the generated mapping to select the particular node. If an assertion was selected, we aim to find an execution path that breaks it as mentioned before; otherwise, we just examine its reachability. That is done by performing the same path exploration algorithm, but starting with an empty path condition.

The exploration is performed asynchronously in the background, as well as the lazy creation of CFGs from the source code. It should not take into account any changes made in the source code after the start of the exploration. To improve the interaction with the user, the results of the exploration are displayed progressively when they appear, instead of waiting to display them all at the end.

To display the results, we utilize a cascade display. When the user clicks on a path in the result list, its control flow appears on the right side of the panel. It is divided by the methods it flows through. After selecting a method, the user can walk through all the statements that are executed there, inspecting their types and result values, if they have any. This is achieved by passing the variable interpretations from the execution model back to the appropriate type models that provide the value to display. If the user selects a line in this table, the corresponding statement in the code is highlighted.

6. Implementation

This chapter dives deeper into the implementation, presenting the dependencies, responsibilities and class diagrams of the particular modules. For each module, the fundamental classes and their interactions are examined.

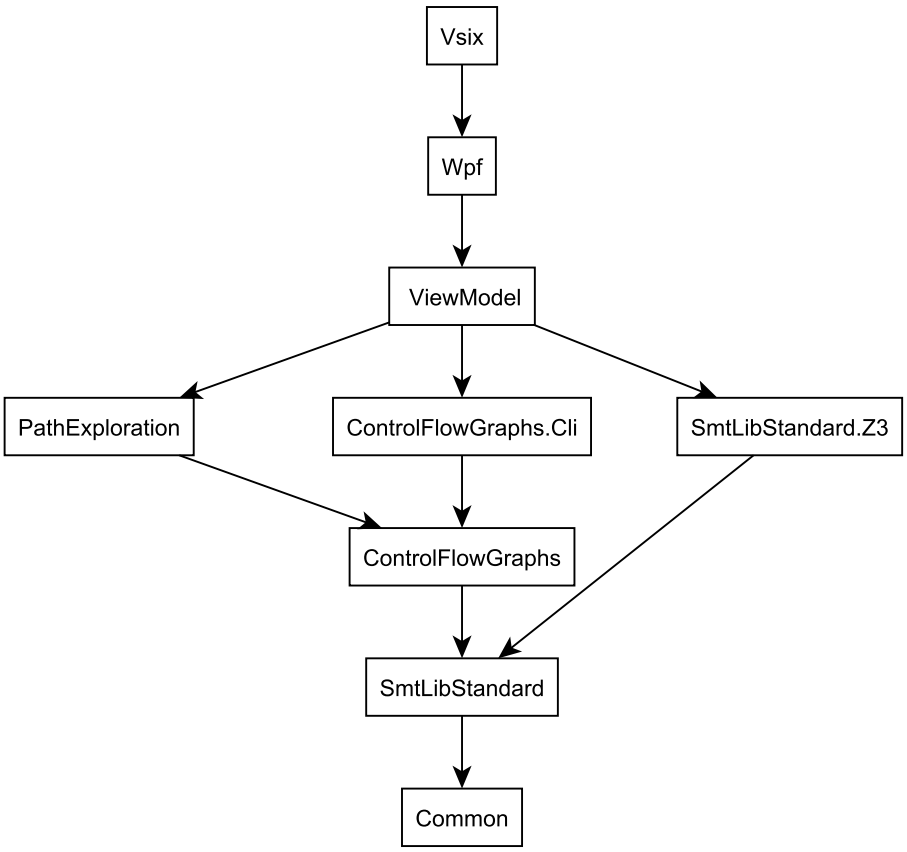

Figure 6.1: Particular modules and their dependencies

On Figure 6.1 are depicted the modules of AskTheCode and the dependencies between them. Transitive dependencies are not shown for the sake of conciseness. We will briefly describe the modules from the bottom up, then we will inspect the architecture of each one in separate sections.

Starting with the module without any dependencies, *Common* is a collection of general purpose interfaces and classes. Among others, it defines a strongly typed way of working with identifiers and overlays, which is heavily used in the other modules.

SmtLibStandard, as its name suggests, contains an implementation of a subset of SMT-LIB standard. It enables to build an expression and work with it in a strongly typed way. An interface for SMT solvers is also defined there. Microsoft Z3 solver [4] wrapper is implemented in separate module *SmtLibStandard.Z3*.

In *ControlFlowGraphs*, data structures to represent CFGs are present. This module also defines an interface to be implemented by libraries that transform source code to a system of CFGs. The implementation for C# is contained in the *ControlFlowGraphs.Cli* module.

The language-agnostic logic of exploring CFGs is contained in the *PathExploration* module. The results it produces are not in a form to be directly understood by the user, they need to be translated back to relate to the language the CFGs were created from.

Among others, this is the job of *ViewModel*. Its name origins from the Model-View-ViewModel pattern [34], where it refers to the last component. The view is represented by a particular user interface in an IDE and the model comprises all the other modules mentioned so far. As we can see, each of it has some dependencies on the functionality provided by other ones. Therefore, ViewModel takes care of properly interconnecting them and making them work in user scenarios.

The implementation of View component is done in *Vsix* module by creating an extension to Microsoft Visual Studio. The module handles the interaction with the IDE, such as text highlighting, context menu entry registering or panel displaying. WPF controls were extracted to a small module called *Wpf* so that they can be used independently. An example of such usage is sample project *StandaloneGui*, which mimics the environment of an IDE in a simple WPF window.

6.1 Common

The most interesting concept in this library is the strongly typed work with identifiers and overlays. The `IId` interface can be implemented by any type, making it an identifier for another type that implements `IIdReferenced<TId>`. Then, for every collection of such type's instances with distinct IDs we can use `IOverlay<TId, TReferenced, TValue>` to associate `TValue` data with them. An example of usage are temporary data that graph algorithms need to store for the nodes they are inspecting, or associating data with immutable objects.

A simple implementation for IDs made up of increasing integers starting from zero is contained in `OrdinalOverlay`. Internally, it just uses the IDs as indices to a list, which is gradually extended according to the need. There is still a room for extensibility, as we can create other types of IDs or overlays in the future.

Another useful interface is `IFreezable`, as it unifies the syntax and semantics of all the types using this pattern. To help the creation of generic collections with a single element, we introduce a lightweight structure named `Singular<T>` and an extensibility method `ToSingular`.

6.2 SmtLibStandard

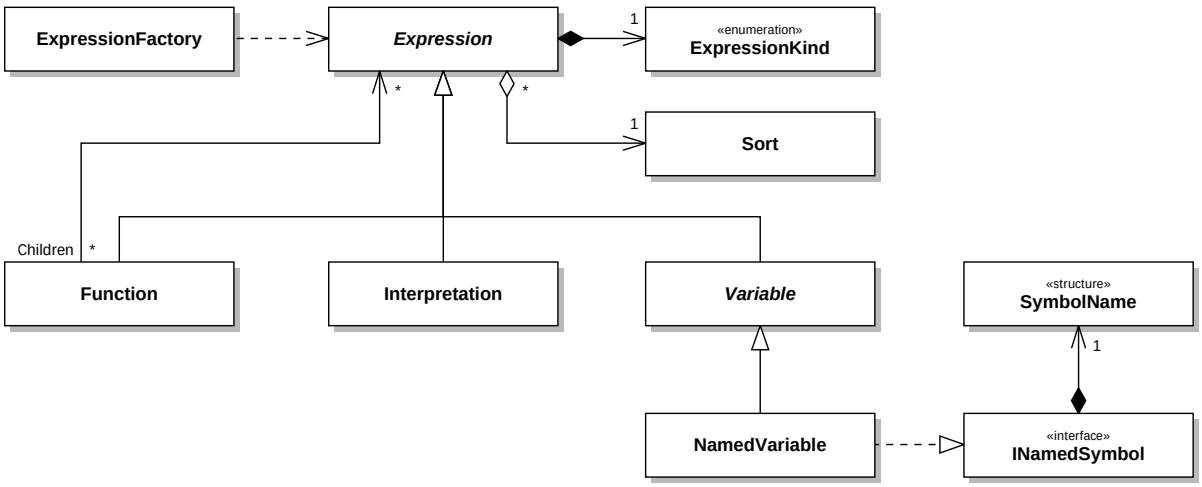

Figure 6.2: Fundamental classes used for storing SMT-LIB standard expressions

The most fundamental entity of this library is the `Expression` abstract class. Among others, it is determined by its sort, kind and children expressions. Figure 6.2 contains the core types strongly related to it.

To denote a type in the world of SMT-LIB, we use the name "sort", hence the `Sort` class. Its instances cannot be created directly; instead, they are managed by the static members of the class. Simple common sorts such as `Bool`, `Int` or `Real` are held as static properties, nested or parametrized ones like arrays or bitvectors can be obtained from static methods. The main reason for keeping the sorts handled globally like that is to maintain their reference equality. As we expect them to be often compared between each other, this approach can contribute both to performance and readability.

Besides the sort, the expression contains also a value from the `ExpressionKind` enumeration. It is used to identify the semantics of the top level token of the expression, possible values are for example multiply operation, boolean negation operation or an interpretation.

Speaking about it, the `Interpretation` class is the simplest implementation of `Expression`. It does not contain any children, only a value from the universe of the corresponding sort.

The only implementation of `Expression` supporting children expressions is the `Function` class. Currently, only the built-in functions are supported, custom functions cannot be defined. Quantifiers are not available, too.

Another interesting inherited class is `Variable`. We decided for this name even though in SMT-LIB standard it is in fact an uninterpreted constant. The reason is that a name such as "Constant" could be confusing for a lot of programmers, as it resembles the semantics of `Interpretation`. `Variable` is abstract itself, enabling everyone to implement their own and use it in expressions.

As we will see later, when using the SMT solver, every variable must be associated with a `SymbolName` structure, representing a unique name in the solver scope. `NamedVariable`, an implementation provided by the library itself, requires this name to be passed upon construction and stores it as a read only field.

None of the mentioned classes have constructors available publicly. Instead, to create expressions, we need to use the `ExpressionFactory` static class. It contains static factory methods for all the supported expression kinds with parameters corresponding to the arity of the particular kind. Moreover, in debug mode, it validates also the sorts of the arguments.

Because of the nature of the problem, all the mentioned subclasses of `Expression` are immutable and the user is advised to keep any `Variable` implementations immutable, too. As such, the expression does not have access to its parent, but it can be shared between multiple ones.

To traverse an expression, the visitor design pattern [35] is implemented using the abstract classes `ExpressionVisitor` and `ExpressionVisitor<TResult>`. From the first one, `ExpressionWalker` is inherited, which dives into the children expressions. The `Visit*` methods of the second one have a return type of `TResult`. Therefore, it is used as a base of `ExpressionRewriter`, which can be utilized to create a new expression from the first one. The used terms and approach were inspired by similar classes operating on the AST in Microsoft Roslyn.

In order to enable the users of the library to use the expressions in a strongly typed way, we introduce *expression handles*. Every such handle is a simple structure named after a certain sort with an expression reference as its only field. The handle can hold reference only to an expression of the particular sort; otherwise, it throws an exception upon construction. Other useful feature is that the handle can simplify the construction

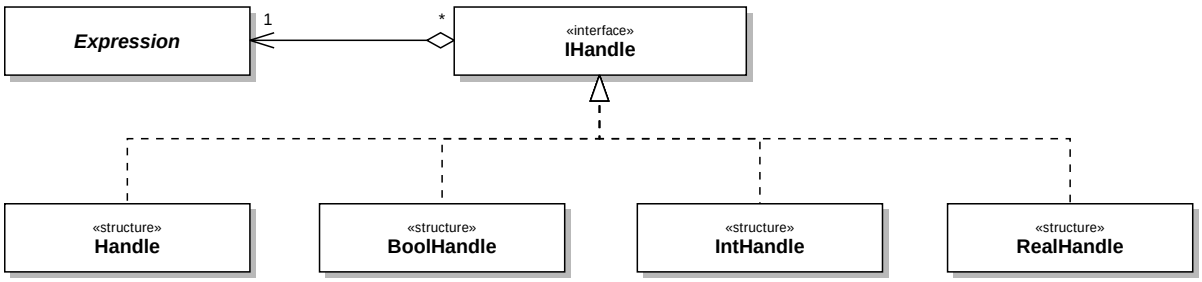

Figure 6.3: Strongly typed expression handle classes

of the expressions of the given type by providing appropriate methods and operators overloads. For example, supposing `a`, `b` and `cond` are handles of sorts `Int` and `Bool`, the following expression in SMT-LIB: `(ite (or cond (> a b)) (+ a b) b)` can be constructed in C# as: `(cond || a > b).IfThenElse(a + b, b)`.

In Figure 6.3 are shown the currently implemented handles. `IHandle` interface, implemented by all the handles, requires the expression reference, simplifying the interactions between the handle types. For example, the mentioned `IfThenElse` method of `BoolHandle` is generic, expecting the both arguments to be of the same type and to implement `IHandle`.

Structure `Handle` serves as an analogy of the base class, as every other handle is implicitly convertible to it. It can be used in an environment, where we work with handles, but do not know the types of some of them in the compile time.

We expect to extend the handles by creating also ones for parametrized sorts, such as arrays or sequences. Such handles could be implemented as generic types, maximizing the compile time checks.

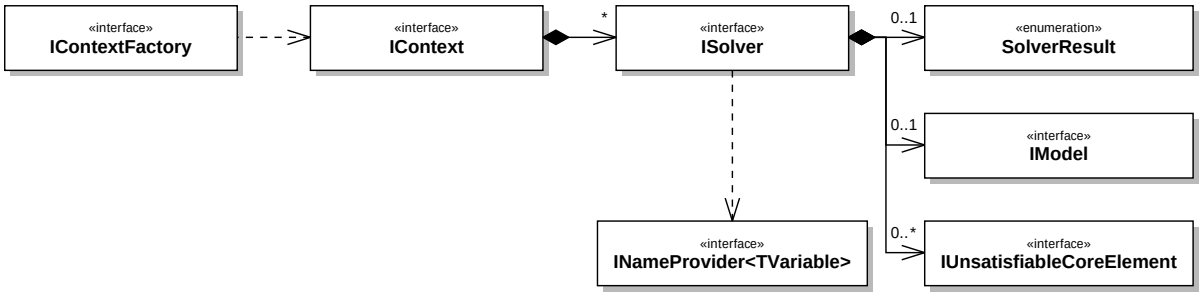

Figure 6.4: Interfaces for a particular SMT-LIB standard solver to implement

The library defines a set of interfaces to be implemented by SMT solvers, as seen in Figure 6.4. The first one, `IContextFactory`, is used to generate solver contexts, implementing `IContext`. Contexts serve as a place where internal structures of the particular solvers are stored, for example, defined variables and expressions. Every context creates and manages a set of solvers, which implement `ISolver`.

The tasks of the solver are to accept assertions, manage their stack and provide the results of the satisfiability checks. The provided assertion expressions must be converted to an implementation specific representation of the particular solver. This is where the `SymbolName` structure comes to place, because it decides the name of the variable the solver uses internally to identify it. If the variables in the expression implement `INamedSymbol`, it is extracted directly from them, as in the case of `NamedVariable`. Otherwise, upon adding the assertion, the caller provides also an implementation of `INameProvider<TVariable>` that assigns the names to the variables upon request. This approach is useful when we have a single expression which we want to assert multiple

times with different variables.

`SolverResult` enumeration can be either `Sat`, `Unsat` or `Unknown`. In the first case, a model class implementing `IModel` can be produced, which provides the interpretations of the variables identified by their `SymbolName`. In the second case, a collection of `IUnsatisfiableCoreElement` implementations can be provided, containing references to the assertions from the unsatisfiable core.

6.3 SmtLibStandard.Z3

The implementation of Z3 is fairly straightforward. The interaction is handled by its .NET API and the particular objects such as `Microsoft.Z3.Solver` are wrapped by classes implementing the `SmtLibStandard` interfaces. To convert the expressions to the Z3 format, there is `ExpressionConverter`, an implementation of the `ExpressionVisitor<TResult>` class.

6.4 ControlFlowGraphs

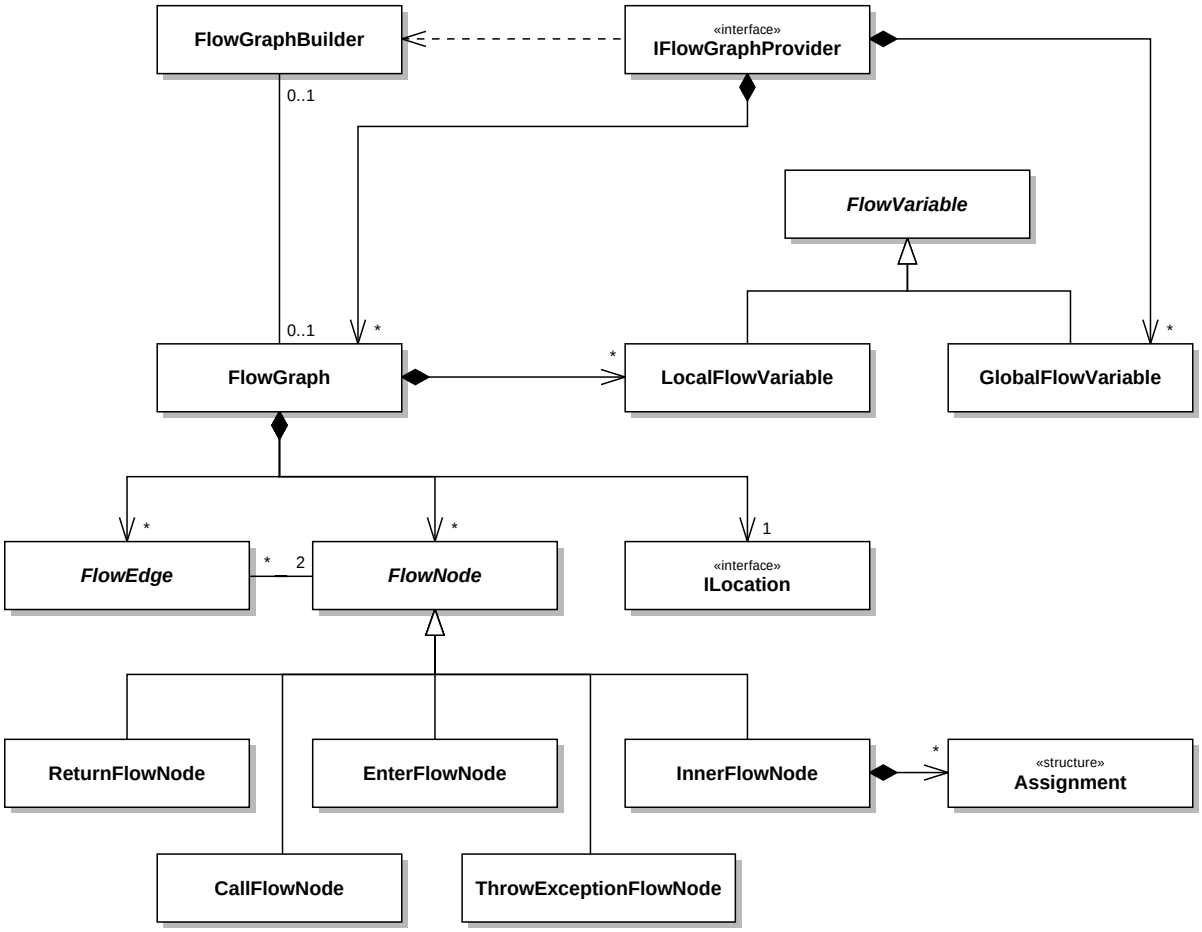

Figure 6.5: Fundamental classes for handling CFGs

The task of this module is to handle the work with programming language independent CFGs. Looking at Figure 6.5, we can see two interfaces to be implemented by a particular CFG generation library. `ILocation` represents the location where the CFG was generated from, e.g. a method definition in the source code. When passed to the

implementation of `IFlowGraphProvider`, it uses `FlowGraphBuilder` to create an instance of `FlowGraph` and returns it, caching the graphs already created.

This class, representing one CFG, implements `IFreezable`, freezing it causes to disconnect the associated builder and become immutable. `FlowGraph` holds collections of local variables, nodes, edges. All of these also keep the reference back to the graph they belong to.

Both local and global variables can be used in expressions among the CFG, but the latter, managed by the graph provider, can appear in more than one graph. In the class diagram, they are represented by `LocalFlowVariable` and `GlobalFlowVariable`, both inherited from abstract `FlowVariable`, which inherits from `Variable`.

The expressions built upon the variables can be used in CFG nodes and edges. As we can see, `FlowNode` is an abstract class, serving as a base for all the CFG node types mentioned in the previous chapter. For example, `InnerFlowNode` holds an immutable collection of `Assignment` structures. Each assignment contains a reference to the variable being assigned to and to the expression representing the new value. `CallFlowNode` and `ThrowExceptionFlowNode`, apart from the argument expressions, hold also the location of the method being called or the constructor of the exception being thrown, respectively. Every `FlowNode` holds the collections of ingoing and outgoing edges and every `FlowEdge`, besides the condition expression, holds the references to the source and the target node.

To be precise, the mentioned semantics works only for `InnerFlowEdge`, which is an implementation of abstract `FlowEdge` to represent edges within a graph. Interprocedural connections between graphs are expressed by `OuterFlowEdge`, instances of which are created on demand and the corresponding border nodes do not know about them.

All the mentioned entities have their IDs, implementing the `IIdReferenced<TId>` interface. The IDs are ordinal, named as `FlowGraphId`, `LocalFlowVariableId` etc. This enables us to use overlays, most of them derived directly from `OrdinalOverlay`, helping in fact only to shorten the type name. `FlowGraphsVariableOverlay<T>` is probably the most interesting one, as it handles both local and global variables, simplifying to associate data with them.

6.5 ControlFlowGraphs.Cli

This module is meant to provide the CFG provider for programming languages adhering to the Common Language Infrastructure (CLI) specification [36]. Currently, only C# is supported, but more languages might be added in the future. The reasons for building the graphs from the source code instead of the Common Intermediate Language (CIL) are that it fits much better into the Microsoft Visual Studio and Roslyn ecosystem and it also looks more promising in the long term.

As we want to help the user orient in the code and provide them visual guides such as CFG viewer or execution path browser, we need to have an access to the detailed information about the source code and its semantics. This is exactly what Roslyn provides, as we can traverse the AST and inspect the underlying symbols and operations. Therefore, when modelling the program behaviour, we have a better control about its mapping to the original code than if we were exploring CIL and its code mapping generated by the compiler. Moreover, we do not have to compile the whole project, Roslyn will provide us the semantics in a lazy manner, saving resources. Furthermore, as we have already mentioned, many advanced features of C# that we might want to support in the future are more easily analysed using Roslyn, because the generated CIL tends to be too complicated.

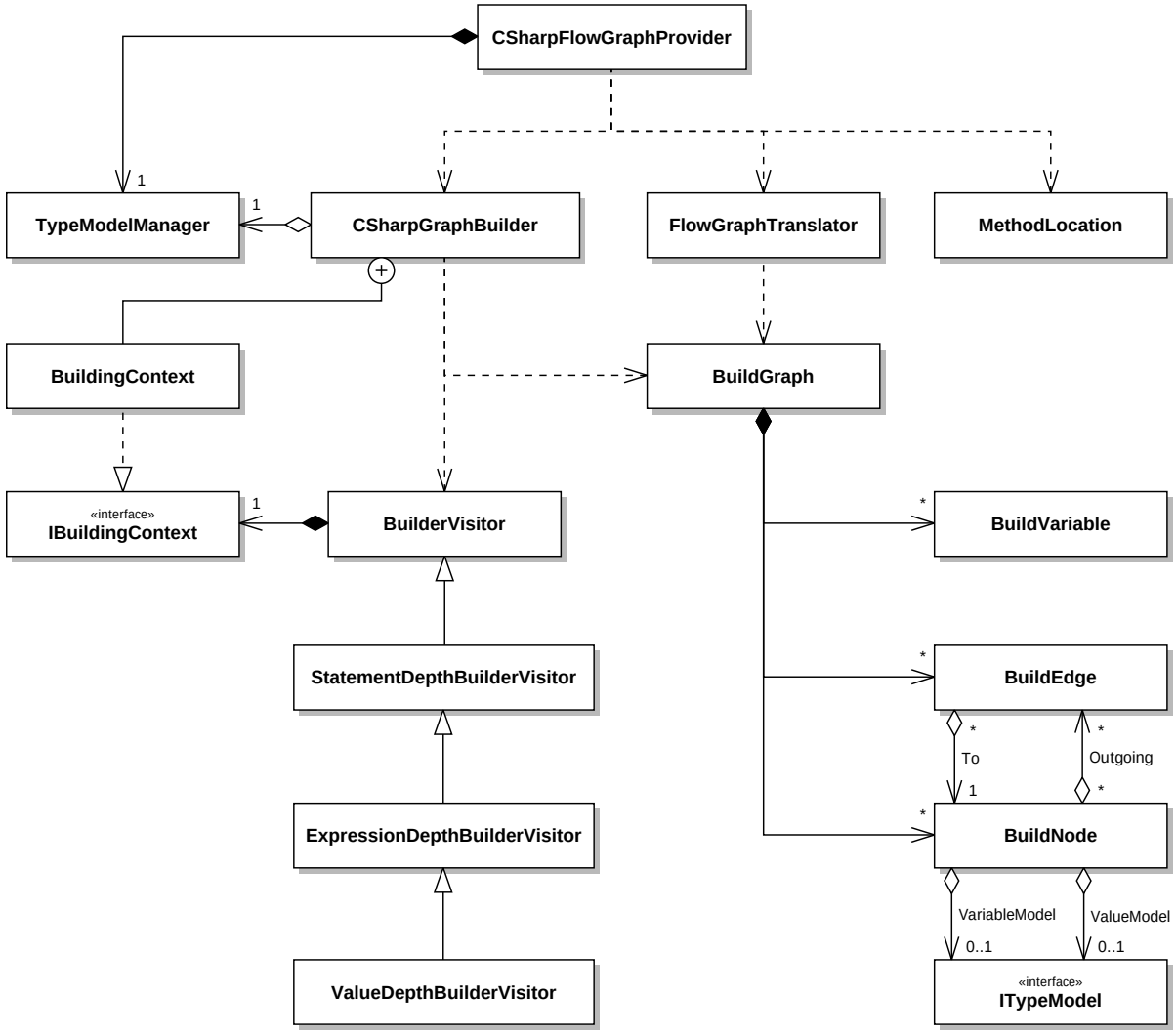

Figure 6.6: Fundamental classes for creating CFGs from C#

The classes handling the purpose of this module are shown in Figure 6.6. The location of a method in the code is stored in `MethodLocation`, which implements `ILocation`. When passed to `CSharpFlowGraphProvider`, it generates the `FlowGraph` and its mapping back to the source code. In this process, it utilizes several internal classes.

`BuildGraph` is an intermediate representation of the CFG, taking into account the specifics of CLI and containing the mapping back to the source code. In contrast to `FlowGraph`, both the graph itself and its nodes, instances of `BuildNode`, are mutable. Every node contains a reference back to the AST node it relates to. It also holds information about what type of node is it, for example inner node or enter node. If any data operation is taking place in the node, it contains the type models of the expression being evaluated and the variable being assigned to. The origin of the type models and their inner workings will be explained below. For now, we can say they provide wrappers around SMT-LIB expressions. To work with these, we need variables; therefore, we provide another implementation of `Variable`, `BuildVariable`. `BuildEdge`, in contrast to `FlowEdge`, refers only to the target node, not to the source one. The reason for this approach is that it makes the graph modification easier. It is also the cause of many-to-many `Outgoing` aggregation between `BuildNode` and `BuildEdge`, because the edges can be shared between several nodes. Unlike `FlowEdge`, it also does not store the whole condition expressions, just the value the `VariableModel` of the source node must have.

The first phase, creating the `BuildGraph`, is handled by `CSharpGraphBuilder`. There are three levels of detail to which a C# method can be analysed, corresponding to the `GraphDepth` enumeration: `Statement`, `Expression` and `Value`. Although we consider this division to be beneficial, currently is all the logic distributed only between the first two. In the future, we want to move the appropriate logic to the `Value` level in order to improve readability and testability.

Although only the last one is used to generate the CFG later, we can use all three for debugging or testing purposes. We provide a sample application, *ControlFlowGraphViewer*, to inspect the graph generation process, giving visual guides about possible implementation errors.

The names of classes responsible for performing the analysis to a particular depth start with the corresponding value from the enumeration and continue with the `-DepthBuilderVisitor` suffix. As we can see in the class diagram, they inherit from each other and the first one inherits from `BuilderVisitor`. This class inherits from Roslyn's `CSharpSyntaxVisitor` and stores `IBuildingContext`, which is used to manipulate the state of the `BuildGraph` being actually built and its builder. For example, it provides the reference to the currently inspected `BuildNode` or enables too look up type models corresponding to the variables in the code.

This interface can be used for testing purposes, but now its only implementation is `BuildingContext`, a private nested class of `CSharpGraphBuilder`. Through it, the visitors recursively expand the nodes corresponding to the particular nodes of the AST, ultimately leading to creating the `BuildGraph` of the desired depth.

It is then send to `FlowGraphTranslator` to be transformed into `FlowGraph` and `DisplayGraph`. The latter is an overlay on top of CFG nodes, mapping them to the locations in the code. Both of these are stored within `CSharpFlowGraphProvider` to satisfy future requests and `BuildGraph` is discarded. The reason is that it operates directly on the nodes of the AST, so keeping a large amount of these graphs alive would prevent them from being garbage collected, thus causing memory leaks.

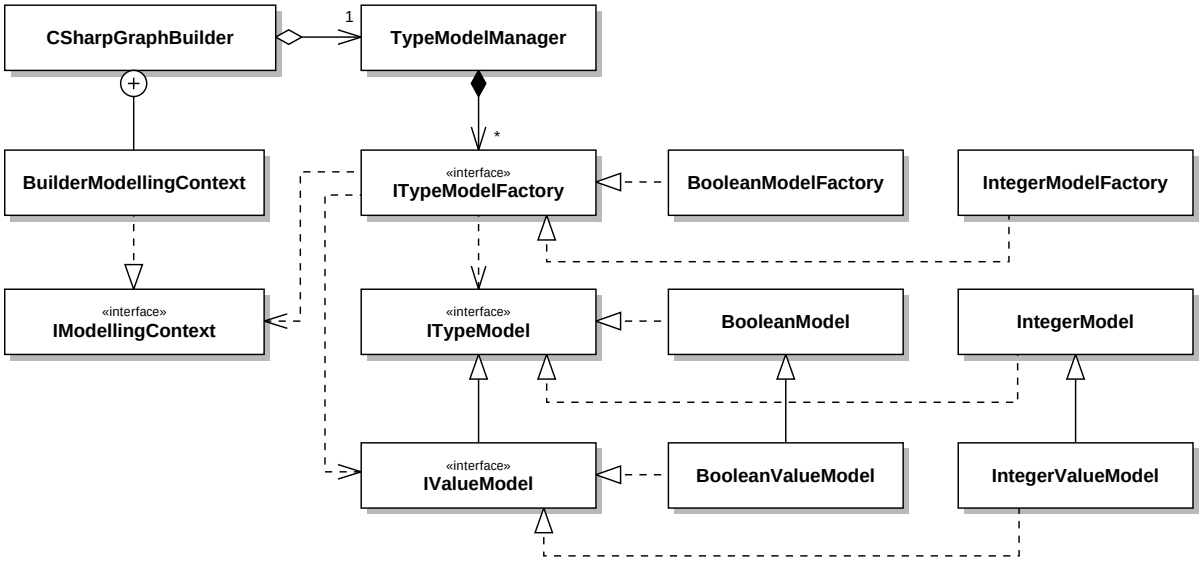

Figure 6.7: Interfaces and classes for handling type models

Having reviewed the CFG generation workflow, we will inspect the entities related to type models. As seen in the previous class diagram, `CSharpGraphBuilder` holds a reference to `TypeModelManager`. This is depicted also in Figure 6.7, together with other interfaces and classes. The manager holds a collection of the implementations of `ITypeModelFactory`. Every factory handles a set of CLI types, for example, in

case of `IntegerModelFactory`, it comprises `int`, `uint`, `long`, `ushort` and other integer types.

It can specify how many expressions and of what sorts it needs to create the corresponding type model. This is useful mainly for creating the variables of the appropriate sorts by the builder and passing them to the factory to obtain an implementation of `ITypeModel`. `IValueModel` is its special case, representing a type model of a particular value, expecting a sequence of `Interpretation` values upon its construction. For instance, `BooleanModelFactory` specifies that `BooleanModel` needs only one variable of `Bool` sort. `BooleanValueModel` can be constructed by passing an interpretation containing either `true` or `false`.

When the type models are created, the builder can use the factory to model their operations. The builder must supply the type models of the operation arguments and an implementation of `IModellingContext` to store the result to. This is a task of `BuilderModellingContext`, its private nested class, which reflects the results of the operation directly in the graph.

6.6 PathExploration

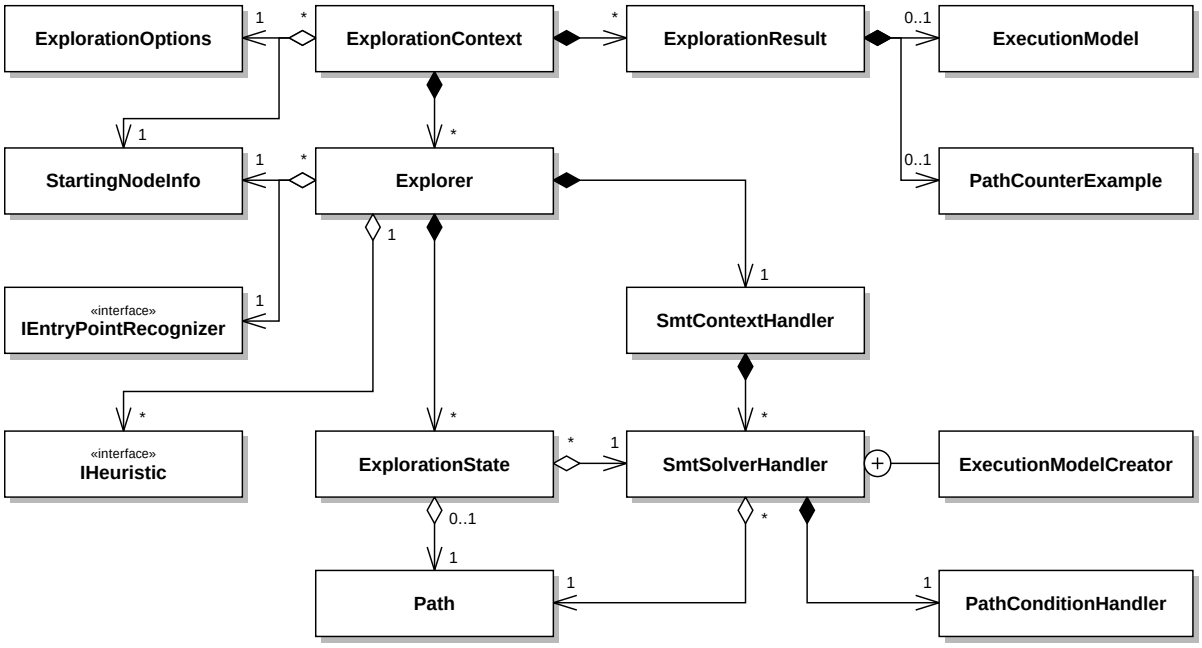

Figure 6.8: Core classes taking care of the exploration

The path exploration algorithm is distributed between several classes, as we can see in Figure 6.8. `ExplorationContext` represents one run of the algorithm. Its inputs are an implementation of `IFlowGraphProvider` and instances of `StartingNodeInfo` and `ExplorationOptions`. The second one contains a reference to the node to start the exploration from and optionally the assertion to verify. The last one contains the heuristics to use in the exploration, they will be further described below. `ExplorationContext` manages a list of the instances of `ExplorationResult`. Because the counterexample creation is not implemented yet, each one can contain only `ExecutionModel`. In the future, also a `PathCounterExample` will be possible.

The class responsible for running the algorithm is `Explorer`. In the future, it is expected that the context creates more than one instance of it, distributes the task among then and runs them in parallel. For now, however, only one explorer is used,

which can be run either synchronously or asynchronously. The starting node instruction and the heuristics obtained from `ExplorationContext` are used as the parameters to Algorithm 1, which we have already explained in the previous chapter. Besides the heuristics, there is also an implementation of `IEntryPointRecognizer`, to inform the explorer about the end of the exploration on a certain position in the code.

`ExplorationState` represents an exploration state in the worklist. It is uniquely identified by its `Path`, because having two states sharing one path would mean we are exploring the same thing twice. `Path` is immutable, we extend it by creating another instance that references it. Every instance holds references to related edge and node in the CFG. It also contains the depth, which corresponds to the number of edges between the starting node and the current one. As mentioned before, paths can also be merged, which leads to an acyclic graph instead of a simple sequence.

To perform the satisfiability checks, we use handlers of `IContext` and `ISolver`. `SmtContextHandler`, besides managing the context, keeps the mapping of versioned CFG variables to the instances of `SymbolName`. `SmtSolverHandler` utilizes `VersionedNameProvider` to store the versions of the variables appearing on the path it is checking. To pass the assertions with the correct versions of the variables to the solver, it uses private nested `VersionedNameProvider`, an implementation of `INameProvider<FlowVariable>`. To create an execution model, `ExecutionModelCreator` is used.

`SmtSolverHandler` keeps the information about the latest path it checked. To check another path, it simply retracts the assertion stack to the last common node of the two paths and pushes the rest of the new path to it. [37] In the beginning of the algorithm, only one state and a corresponding solver handle is created; however, when expanding the states, they can possibly share the same one. Therefore, multiple states can reference the same solver handler, but it is always related only to one path it currently checked. To decide whether the solver handler should be shared or cloned, we introduce `ISmtHeuristic` as another heuristic to parametrize the algorithm.

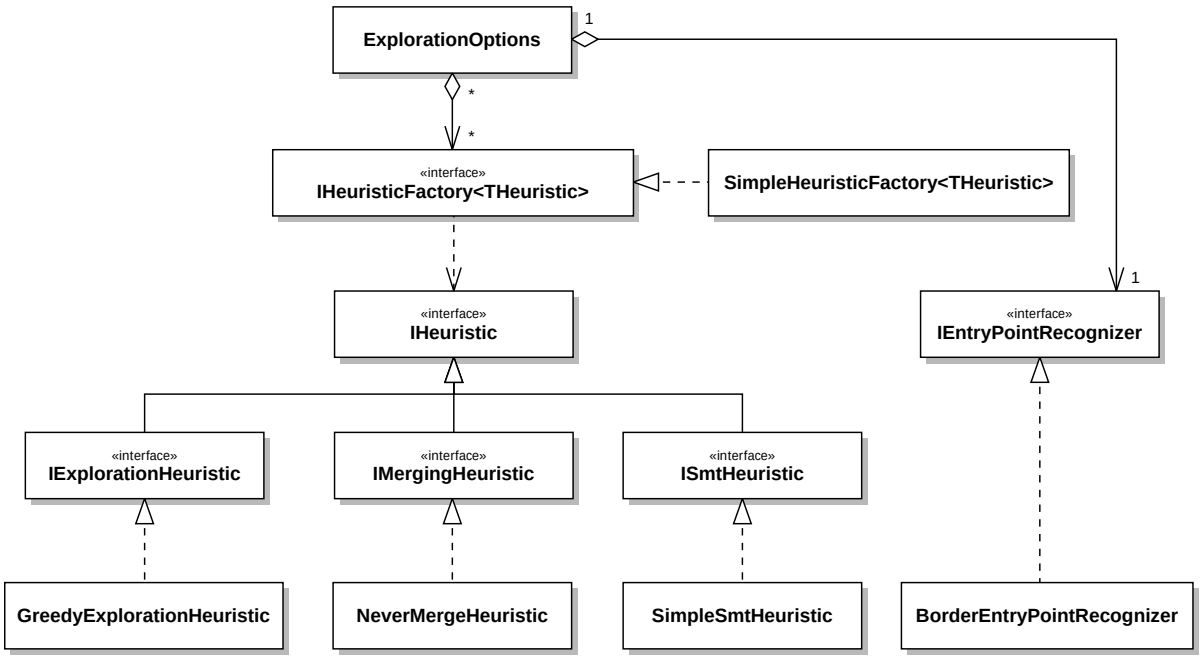

Figure 6.9: Interfaces and classes handling the path exploration heuristics, including straightforward implementations

The system of heuristics is described in Figure 6.9. The `ExplorationOptions`

class does not contain the heuristics itself, because they are expected to maintain an inner state related to the current exploration. Instead, it holds references to the strongly typed implementations of `IHeuristicFactory<THeuristic>`. These factories can be shared between the sets of options. If the factory does not perform any logic itself, `SimpleHeuristicFactory<THeuristic>` can be used, which only passes the `Explorer` reference to the heuristic being created.

The particular interfaces correspond to the heuristics we have already described. In the class diagram, also their straightforward implementations are depicted. Starting from `IExplorationHeuristic`, `GreedyExplorationHeuristic` picks the next state randomly and always agrees with following a certain edge. `NeverMergeHeuristic`, according to its name, always denies merging; therefore, it also does not instruct the solver handler to expect it. `SimpleSmtHeuristic` makes the solver handler to never clone itself and to perform the satisfiability check on every node.

As `IFinalNodeRecognizer` is not expected to keep any inner state regarding the current exploration, it does not have to be created using a factory. Therefore, its implementation is inserted directly to `ExplorationOptions`.
`BorderFinalNodeRecognizer` is a simple implementation that stops the exploration at the enter node of the CFG, preventing the interprocedural checking.

6.7 ViewModel

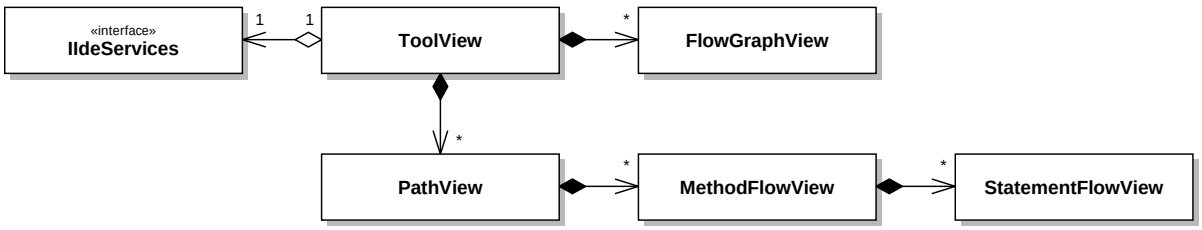

Figure 6.10: Class diagram of ViewModel

`ToolView` represents the state of the main panel, managing all the other views. To interact with the IDE, it uses an instance of `IIdeServices`, which provides access to text selection, highlighting etc. Other types of interaction are done via binding to the `*View` classes, utilizing `INotifyPropertyChanged` and `ICommand` interfaces. When the user commands to explore the code, `ToolView` is responsible for interconnecting the underlying modules, starting the asynchronous exploration and supplying the results to the appropriate classes.

As we can see, `PathView`, `MethodFlowView` and `StatementFlowView` have tree-like logic, which aims to simplify the provision of the results to the user. When the user selects a statement, it gets highlighted in the code.

The graph is built according to the actual CFG and its mapping to the source code.

To try the path exploration in a user interface without having to use an IDE, we have created a simple application based on Roslyn, *StandaloneGui*. It provides a user interface created using Windows Presentation Foundation (WPF), enabling to start the exploration and see its result, just as in the case of the IDE extension. Of course, it cannot be used as an IDE, it does not even support code editing or syntax highlighting. However, it supports one feature that is not available in the Visual Studio Extension, namely CFG displaying. To use it, right click on a method in the code and click on the corresponding command. It is rendered using Microsoft Automatic Graph Layout

(MSAGL) library [38, 39] and handled by `FlowGraphView`. Note that it is an unstable feature at this moment.

6.8 Vsix

As we have handled all the logic in the preceding modules, the implementation of Microsoft Visual Studio extension is quite straightforward. The most of the features were simply registered to the appropriate hooks of the IDE, the only advanced thing was the handling of code highlighting.

In the most of the documentation samples, code highlighters in Microsoft Visual Studio are self-contained, gathering all the needed information from the document they are highlighting the code in. In our case, however, the highlighting is driven by an event not related to it, but to another panel in the IDE. Therefore, we developed `HighlightingService`, a class that manages all the highlighters of the opened documents and redirects the highlighting requests to them as needed.

Part III

Evaluation

7. Experiment Design

We have already described the inner workings of AskTheCode and its relation to the existing tools for code analysis. In this part, we will evaluate to what extent AskTheCode fulfils its purpose and whether it brings any newly added value for the user. This chapter enlightens the experiment setup, the following chapters will present the results and interpret them.

Although the final goal of AskTheCode is to be practically useful, there is still a lot of work ahead, such as modelling objects on the heap, collections etc. As we can evaluate only the analysis of static methods[1] and operations on basic types, it is problematic to find any *real life* programs as representative examples. Therefore, some of the created examples were tailored to assess the strengths and weaknesses of our solution, others come from various works related to symbolic execution. [10, 40, 41, 42]

In every example, one or more assertions are present. We use AskTheCode to assess their validity one by one. To compare our approach with existing broadly used technologies for C#, Microsoft Pex and Code Contracts static checker will be used to perform the same task. FxCop, regrettably, cannot take part in the evaluation, because its reasoning is aimed mainly at handling of null values among references, proper using of `IDisposable` and other advanced features. Therefore, it was unable to find any relevant errors in our examples.

Obviously, Pex was not meant to verify valid assertions, but it can generate inputs to break the invalid ones. Similarly, Code Contracts static checker can help by proving a valid assertion, but it is hard to judge whether it was actually useful in the case of an invalid one. In every such situation, we will assess whether the information provided by it was actually useful, possibly helping the user to discover the problem in the code. For example, just stating that the assertion was unproven or directly passing it to the preconditions of the method will not be considered as the right answer.

Both of the mentioned technologies are used to process all the assertions at once; therefore, comparing the amount of time spent with AskTheCode would be problematic. Moreover, it would be complicated to measure the exact running time of the current version of Pex, integrated to Visual Studio, as it has no command line interface. As a result, we do not care about the time consumed by these tools, apart from constraining it by the reasonable maximum value of 30 seconds per method.

Example group	Ifs	Loops	Valid assertions	Invalid assertions
IntegerArithmetic [10]	5	0	4	5
TriangleClassification [40]	10	0	1	5
BankAccount [41]	4	1	0	3
Loops [42]	5	7	4	6

Figure 7.1: Overview of the example programs used for evaluation

In Figure 7.1 are summarized the basic statistical data about the groups of example. Every group corresponds to a class in the *EvaluationTests* project. The best way to

[1] To be precise, AskTheCode does not reject to explore instance methods, but cannot model the access to the instance members.

acquaint oneself with the examples is to inspect the source code and its comments. Their short summaries follow:

IntegerArithmetic contains various straightforward methods covering linear and non-linear integer arithmetic. Some of these are aimed the implementation of 32-bit integers and their operations. For example, in `AlmostAbsoluteValue`, the counter example depends on the fact that `-int.MinValue` is not a positive number. To evaluate interprocedural approach, we validate axioms of 2D vector dot product, implemented as a separate method.

TriangleClassification is an implementation of a classic benchmarking problem for test input generators, sometimes called *Trityp*. The algorithm accepts the side lengths of a triangle and returns its type: scalene, isosceles, equilateral, or invalid. Although the algorithm contains no loops, its nesting and data flow are so complicated that its correctness is not immediately apparent. Besides the original algorithm, also its five mutants [40] with various introduced errors are included.

BankAccount simulates a simple banking application by keeping the account balance and performing a sequence of deposit and withdraw operations. The particular sequence is modelled by calling indeterministic methods in a loop. The code contains a set of statements falsely marked as unreachable.

As its name suggests, *Loops* presents various type of loops to demonstrate both the benefits and limitations of AskTheCode. In the `ConcolicInefficientLoop` method, there is a valid assertion that can be verified without even reaching the loop, showing the clear advantage of backward symbolic execution against the forward one. The remaining methods are more complicated, as we need to traverse the loops multiple times, while some of them contain branching to make the problem even harder.

8. Results

The purpose of this chapter is to present the results of the proposed experiment. The summary of the assertion verification results is shown in Figure 8.1. For every program and tool, the counts of proved and refuted assertions are listed. In the case of contracts, if adding any hints to the checker helped to increase the score, it is written in parentheses.

	AskTheCode	Code Contracts	Microsoft Pex
IntegerArithmetic	4 / 4	2 (5) / 4	0 / 4
TriangleClassification	1 / 1	0 / 1	0 / 1
Loops	2 / 4	2 (3) / 4	0 / 4

(a) Proved assertions

	AskTheCode	Code Contracts	Microsoft Pex
IntegerArithmetic	2 / 5	4 / 5	4 / 5
TriangleClassification	5 / 5	3 / 5	5 / 5
BankAccount	3 / 3	0 / 3	3 / 3
Loops	4 / 6	0 / 6	5 / 6

(b) Refuted assertions

Figure 8.1: Results of the evaluation in terms of proved and refuted assertions

This happened in the case of the interprocedural examples from *IntegerArithmetic*. The checker was not able to reason about the dot product operation in the axioms, until we duplicated the return formula to its postconditions. Other tasks were handled well by it, as well as by Pex. AskTheCode managed to solve all the examples except for those requiring the exact modelling of the operations with respect to the particular bits.

TriangleClassification examples were successfully solved both by AskTheCode and Pex. In three of the mutants, the contract checker spotted branching conditions evaluating always to the same value, which would have potentially helped the user to find the errors. However, it was neither able to prove the semantics of the original operation, nor to find any particular violations in the case of the mutants.

BankAccount was successfully simulated by both AskTheCode and Pex, as both provided the series of operations of how to reach the target statements. The contract checker was not able to prove whether are they reachable or not.

AskTheCode was unable to prove the valid assertions in the complicated examples of *Loops*; however, it successfully verified two assertions positioned after a loop, because the exploration ended before even reaching it. These assertions were verified also by the contract checker. Furthermore, it also verified an assertion in the body of the `SimpleLoop` method. Regarding the refutation of the invalid assertions, it again only stated that they might be reachable. Pex was able to find counterexamples in all the methods except for the `SimpleLoop`. AskTheCode refuted the same assertions as Pex apart from the one in the `LoopedModulo` method.

9. Discussion

The results presented earlier will be interpreted in this chapter. We will reveal the weaknesses and strengths of AskTheCode in the first section and then suggest some improvements for the future.

9.1 Results Interpretation

AskTheCode was unable to properly model some of the arithmetic operations, because it uses simple models, which do not respect the particular low level aspects of how the numbers are stored. Pex uses more advanced bit-vector theory, enabling it to model the operations in a precise way.

Another thing Pex naturally handles better are calls to external methods. As it operates on a runtime of a program, there is no problem in executing them and observing their result. AskTheCode handles everything in a symbolic way; therefore, it cannot reason about any operations that are not modelled. The contract checker can at least utilize the preconditions and postconditions, if they are provided.

Once cycles are involved, AskTheCode is generally unable to prove the assertion. The reason is that it cannot stop unwinding, as it does not know whether is not any future iteration going break the assertion. As a result, it continues to loop until the timeout is reached. For comparison, the contract checker is able to reason about cycles, if it has enough information.

In the case of data dependent cycles, Pex takes a similar approach as AskTheCode, only from a different direction. Working with a running program is more effective than just simulating it in a symbolic way; therefore, in a given amount of time, it is able to explore much more iterations than AskTheCode.

AskTheCode is able to work in an interprocedural way, which has both positive and negative implications. It is able to reason about the exact semantics of the called method, preventing false positives. However, in the case of larger programs, it might complicate the analysis by increasing the path condition and by contributing to path explosion.

AskTheCode seems to generally perform best in the case of examples with an extensive branching, but without loops. Such programs might not be suitable for the contract checker, but Pex is able to handle them, too. The added value of AskTheCode in this case is to ensure the user that the assertion is valid.

If an assertion unprovable by the contract checker is hidden in a system of complicated loops and the statements securing its validity are close to it, it is also an ideal situation of using AskTheCode. Because it explores the control flow backwards, it will prove the assertion before even reaching the loops, thus saving time needed to explore them.

9.2 Future Work

We have already assessed the current state of AskTheCode. Now, we will present the possible ideas how to alleviate the weaknesses and enhance the strengths.

As we can see, it can already help the user in several cases and it has a potential to become a generally useful tool. To achieve that, however, it must be able to model fundamental features of common programs first. We need to simulate the work with the objects stored in the heap and their references, handle instances of custom classes and their fields etc.

To model these, we will probably use CFG global variables and the theory of functional arrays, as it is done in Pex [25]. We can also take advantage of other theories supported by Z3. For example, bit-vectors are a way to properly model numeric values and operations. To simulate working with strings and collections, the theory of sequences might be used.

To reduce the problem with the cycles, proper control and data flow analysis should be performed and its result reflected in the exploration. For example, if we know that the loop does not affect the assertion we are verifying and it is not infinite, we can safely skip it. Or, some loops might be transformed to a finite sequence of operations.

In such analysis, it might be useful to cooperate with the contract checker, as it can be capable to provide us some hints. Furthermore, in the case of larger programs, reasonable combination of code contracts and path exploration might help to fight path explosion while preserving the soundness.

The ideas for future development mentioned earlier are still valid, namely: *interactive exploration*, *code changes suggestions* and *support of multiple programming languages*.

Conclusion

The main goal of the thesis was to create AskTheCode, a novel static code analysis tool for C#, to help the user understand the behaviour of the program, fix errors and prevent creating new ones. After the analysis of the current state-of-the-art techniques and tools, we have specified its main requirements so that it brings an added value, instead of duplicating their functionality. Firstly, we wanted to minimize the prerequisites, such as unit tests setup or code annotating. Secondly, the approach will be demand-driven, instead of analysing all the potential problems in the program. Thirdly, the analysis cannot be limited to one procedure, it must take into account the interprocedural control flow. Lastly, the result of the verification must be provided in an appropriate detail, so that it truly helps the user.

As a result of the requirements, we have implemented AskTheCode as a Microsoft Visual Studio extension capable of verifying a code assertion selected by the user. The used technique is backward symbolic execution, to check the satisfiability of the path condition we utilize Microsoft Z3 SMT solver. The results are displayed in an interactive, detailed way. According to the thesis assignment, only C# basic value types and static methods are supported in the current version, with the possibility to add more features in the future.

Such limited extent is an obstacle for properly assessing whether AskTheCode could be practically useful in the everyday life of a programmer. However, the results of the evaluation of the current state seem promising, showing it has a potential to achieve that. Obviously, we cannot expect it to become a "silver bullet" and overwhelm well-established tools like Microsoft Pex or Code Contracts static checker. Despite that, in some specific cases, AskTheCode was able to find the solution, whereas the two previously mentioned were not. Moreover, even in the cases where the same information about the assertion could have been obtained by using them, AskTheCode can provide more information to the user.

As we can see, the main goal of the thesis was successfully accomplished, because AskTheCode fulfils the requirements and does not duplicate the functionality of existing tools for C#. Thanks to its loosely coupled architecture, it can be easily gradually extended and serve as a base for a future dissertation.

Bibliography

[1] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[2] Amit Goel, Sava Krstić, and Alexander Fuchs. Deciding array formulas with frugal axiom instantiation. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 12–17, New York, NY, USA, 2008. ACM.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Nikolaj Bjørner. *Engineering Theories with Z3*, pages 4–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[6] E. M. Clarke. *Model checking.* Cambridge : MIT Press, c1999, 1999.

[7] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[8] R.L. Sites. Branch resolution via backward symbolic execution, June 27 1995. US Patent 5,428,786.

[9] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.

[10] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Västerås, Sweden, September 15-19 2014. ACM.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[12] A Venet. Automatic analysis of pointer aliasing for untyped programs. *SCIENCE OF COMPUTER PROGRAMMING*, 35(2-3):223 – 248, 1999.

[13] A Venet. Nonuniform alias analysis of recursive data structures and arrays. *STATIC ANALYSIS, PROCEEDINGS*, 2477:36 – 51, 2002.

[14] Manuel Montenegro, Ricardo Peña, and Clara Segura. Shape analysis in a functional language by using regular languages. *Science of Computer Programming*, 111(Part 1):51 – 78, 2015.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[16] KRM Leino and F Logozzo. Loop invariants on demand. *PROGRAMMING LANGUAGES AND SYSTEMS, PROCEEDINGS*, 3780:119 – 134, 2005.

[17] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[18] Nicu G. Fruja. The correctness of the definite assignment analysis in c. *The Journal of Object Technology*, 3:29–52, 2004.

[19] John Robbins. Bugslayer: Bad code? fxcop to the rescue. *MSDN*, 19(6):137, 2004.

[20] Michael Sync. Status of fxcop/code analysis! – is fxcop still being maintained?, 2014.

[21] TED NEWARD and JOE HUMMEL. Rise of roslyn. *MSDN*, 29(11):70, 2014.

[22] Larry Golding. RulesInventory.csv, 2016.

[23] Soma Somasegar. Devlabs: Code contracts for .net, 2009.

[24] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, FoVeOOS'10, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] Pratap Lakshman Tao Xie, Nikolai Tillmann. Advances in unit testing: Theory and practice. In *Proc. 38th International Conference on Software Engineering (ICSE 2016)*, May 2016.

[27] Nasa jpl selects coverity to find defects in curiosity mission-critical flight software. *Military & Aerospace Electronics*, 23(11):28, 2012.

[28] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[29] Johann Schumann and Stefan-Alexander Schneider. *Automated Testcase Generation for Numerical Support Functions in Embedded Systems*, pages 252–257. Springer International Publishing, Cham, 2014.

[30] Coverity® scan open source report 2014, 2015.

[31] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[32] Milen Dzhumerov. Symbolic execution of distributed software, 2011.

[33] Jennifer Greene and Andrew Stellman. *Applied Software Project Management*. O'Reilly, first edition, 2005.

[34] Josh Smith. Patterns - wpf apps with the model-view-viewmodel design pattern, 2009.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[36] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. Geneva, Switzerland, 5 edition, December 2010.

[37] T. (1) Liu, M. (1) Taghdiri, M. (2) Araújo, and M. (2) d'Amorim. *A comparative study of incremental constraint solving approaches in symbolic execution.*, volume 8855 of *Lecture Notes in Computer Science*. Springer Verlag, (1)Karlsruhe Institute of Technology, 2014.

[38] L. Nachmanson, G. Robertson, and B. Lee. Layered graph layouts with a given aspect ratio, April 26 2011. US Patent 7,932,907.

[39] Lev Nachmanson and Lynn Powers. Microsoft automatic graph layout (MSAGL), 2016.

[40] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. *Formal Concept Analysis Enhances Fault Localization in Software*, pages 273–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[41] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 297–306, September 2008.

[42] Florence Charreteur and Arnaud Gotlieb. Constraint-based test input generation for java bytecode. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 131–140, 2010.

Attachments

AskTheCode User Documentation

Here we describe the AskTheCode extension of Microsoft Visual Studio from the user's perspective.

Installation

Make sure you meet the following system requirements prior to the installation:

- Microsoft Windows 7 SP 1 or newer
- Microsoft .NET Framework 4.6.1. or newer
- Microsoft Visual Studio 2015 Update 2 or 3[1]
- 15 MB of free hard drive space

To perform the installation, follow these steps:

1. Open the `bin/AskTheCode.vsix` file on the enclosed CD. VSIX Installer should be launched by this action.
2. After the installer loads all the information about the package, keep the version of Visual Studio checked and click on the *Install* button.
3. Wait until the installer successfully finishes.

Usage

For the first usage of AskTheCode we recommend copying the contents of the enclosed CD to a local folder, so that one can inspect and edit the code. The `test/inputs/csharp/Samples/Samples.csproj` project is intended to illustrate the current capabilities of the tool on a collection of simple examples. Another project with useful samples is `test/inputs/csharp/ EvaluationTests/EvaluationTests.csproj`, which was used for the evaluation of the tool. When creating a custom code to check, below is a list of the current limitations of AskTheCode to keep in mind.

To start using the installed extension in Visual Studio, open the *Tools* menu and click on the *AskTheCode* item. At the bottom of the screen, a tool panel as in Figure 9.1 will appear. The panel is divided into two parts: the left part controls the exploration as a whole, while the right part enables to browse the found execution paths in an interactive way.

At the top of the left part we can see two buttons for starting the exploration. Prior to using one of them, the caret of the currently opened source code file must be on a location in the code that is to be inspected. The purpose of the *Check reachability* button is to verify whether there exist any execution paths from entry points leading to the location. As the entry points are considered all public methods. *Check correctness* can be used to verify the selected assertion by searching for the execution paths that break it.

[1] The previous versions of 2015 should work, too. However, only the mentioned versions were tested.

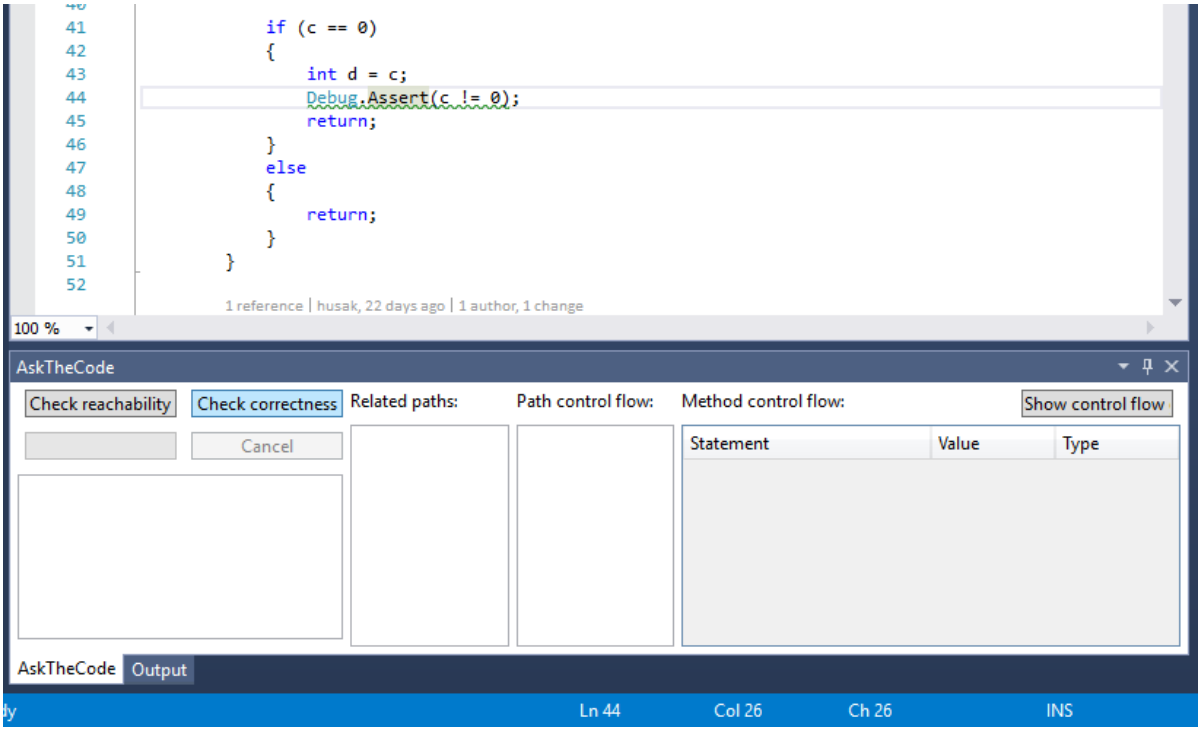

Figure 9.1: User interface of AskTheCode in Visual Studio 2015

After clicking on one of these buttons, the exploration starts. The progress bar underneath the buttons starts moving and a button to cancel the exploration appears. Also, in the *Messages* section at the bottom of the panel we can see the notification about the started exploration. During the exploration, other messages may arise, such as warning or errors. In the end, the message about the overall result is displayed. Note that in the case of simple scenarios the whole scenario can happen so quickly that we are unable to spot the progress bar moving and we only see the messages.

As we have mentioned earlier, the right part of the panel contains the found execution paths, which we can browse in a hierarchical manner. After selecting a path from the first list, we can inspect its flow through the various methods in the *Path flow control* list. The *Method control flow* list then contains the particular statements concerning the flow in the given method. For each statement, its value and type are provided. Selecting a statement will highlight it in the text editor.

Limitations

In the current state, AskTheCode supports only a subset of C# language constructs. It is not able to model access to instance members of a class or structure; therefore, only static methods can be inspected soundly. Also, parameters can be passed only by value, not by reference. Regarding the control flow, the following statements are supported:

- `return`
- `throw new` - it cannot throw an exception created beforehand
- `if`, `else if`, `else`
- `while`
- `&&` and ||

The only supported types are `bool` and all the integral types: `sbyte`, `byte`, `char`, `short`, `ushort`, `int`, `uint`, `long` a `ulong`. There is one unbounded integer model for all the integral types, hence the overflows are not taken into consideration. Although this fact violates the absolute soundness of the tool, in terms of its general usability and performance it is an acceptable trade-off.

For `bool`, `!`, `==`, `!=`, `&`, `&&`, | and || operations are supported. Integral types support `>`, `<`, `>=`, `<=`, `+`, `-`, `*`, `%` and `/`. The division is not modelled precisely, because it does not reflect the throwing of an exception if the second operand is zero.

When loops or recursion are involved in the code being inspected, AskTheCode does not guarantee to explore all the possible paths. In most of the cases it will get stuck in an infinite loop. Although it may deliver certain results during the exploration, there is no guarantee that the list is exhaustive.

CD Content

The top level folder structure of the enclosed CD is summarized in the table below:

Folder	Contents
`bin`	AskTheCode VSIX package
`doc`	Text of this thesis
`lib`	External libraries referenced directly
`samples`	Source code of sample programs built upon AskTheCode libraries
`src`	Source code of AskTheCode
`tests`	Unit tests of the low-level AskTheCode modules