Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring / ~~Autumn~~ semester, 2019. Open/~~Confidential~~
Author: Nicolas Fløysvik	……Nicolas Fløysvik…… (signature of author)
Programme coordinator: Hein Meling Supervisor(s): Hein Meling	
Title of master's thesis: Using domain restricted types to improve code correctness	
Credits: 30	
Keywords: Domain restrictions, Formal specifications, symbolic execution, Rolsyn analyzer,	Number of pages: 75…… + supplemental material/other: 0…… Stavanger,15/06/2019…… date/year

Domain Restricted Types for Improved Code Correctness

Nicolas Fløysvik
University of Stavanger

Supervised by: Professor Hein Meling
University of Stavanger

June 2019

Abstract

ReDi is a new static analysis tool for improving code correctness. It targets the C# language and is a .NET Roslyn live analyzer providing live analysis feedback to the developers using it. ReDi uses principles from formal specification and symbolic execution to implement methods for performing domain restriction on variables, parameters, and return values. A domain restriction is an invariant implemented as a check function, that can be applied to variables utilizing an annotation referring to the check method.

ReDi can also help to prevent runtime exceptions caused by null pointers. ReDi can prevent null exceptions by integrating nullability into the domain of the variables, making it feasible for ReDi to statically keep track of null, and detecting variables that may be null when used. ReDi shows promising results with finding inconsistencies and faults in some programming projects, the open source CoreWiki project by Jeff Fritz and several web service API projects for services offered by Innovation Norway. Three main types of faults where found, input validation, internal API validation, and nullability faults.

Acknowledgements

I would like to thank Professor Hein Meling for supervising the thesis and providing valuable feedback. I would also like to thank Innovation Norway and Bouvet for allowing me to perform some analysis on their code projects.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are many techniques and tools which aims to improve code correctness. The most common tools we have access to are the type system of a programming language and different kind of analyzers and linters. More advanced forms for code analysis includes static code analyzers, deadlock and race detection, and symbolic execution. These tools try to analyze the code and report errors for problematic code to prevent that the software crashes, deadlocks, or operates incorrectly at runtime.

An alternative to checking already written code is to create a model first, then write the code afterward. A model is a specification which describes an object or a system. It could include properties of the model, but also rules specifying the legal values for the properties. In most programming languages, the kinds of rules that a developer can apply to the different properties are quite limited. Programming languages usually restrict these rules to simple and broadly applicable types such as strings, integers, booleans, and floating point values or other structures contained several of the mentioned types.

Software systems increase in complexity since developers add more features to the system, and it can reach a point where a single person can not easily understand the entire code-base anymore. Not only large applications but also in smaller applications can it be hard to remember to test for all the different scenarios and edge cases. Researchers and developers have created several methods to make the software correct and secure by design. Some of these methods are *formal specification* methods such as *design by contract* and *formal modeling languages*, and *symbolic execution*.

There are many tools aimed at improving the correctness of software, which includes Eiffel, TLA+, Dafny, KLEE, and others, but most of them require the developers to learn a new language. It also makes it necessary to have two different implementations, one for modeling the system, and the other for the implementation. Dafny and Eiffel are exceptions to the rule since they are fully

fledged programming languages but requires a separate syntax for defining pre and post-conditions, and invariants.

Another approach could be to implement the model checkers directly into the language itself since programming languages usually have many ways to check variable values. If we combine symbolic execution with model checking, it is possible to define the model checking rules with using symbolic execution on if statements and have them verified with a model checker. The thing missing from mixing the two techniques is a way of linking the variables to the checks, which we can accomplish by using comments or a language feature such as annotation. We do not need to add any new syntax when we combine model checking and symbolic execution and only uses comments or language features for the analysis. Without the need for new syntax in the host language, we also avoid needing any pre-compilation steps or specialized software for the analyzer to function correctly.

In this thesis, we introduce a new static analysis tool, **Re**stricted **D**oma**i**n, ReDi, and technique such as domain restricting type for model checking of languages without needing extra syntax. ReDi uses techniques from several code analysis methods, including symbolic execution, formal specification, and static code analysis. It requires one extra library for using the new constructs introduced, but the helper library only contains annotations for ReDi to know what part of the code to analyze. We also implement type invariants to verify that all the developer uses the code in a way which the model permits. Developers also define the model in the source code itself, providing additional documentation during development and review. The additional documentation provided by the model would also not be outdated since it directly affects the source code.

The Integrated Development Environment (IDE) can provide better analysis and code help if the ReDi is active in the IDE. If ReDi is not active in the IDE, then ReDi cannot enforce the compile-time checks, but the code still uses the checks at runtime if they are percent.

The thesis also uses parts from symbolic execution to determine the scope and domain for the different variables, and ReDi uses the calculated domain as the primary way of defining the models. Developers needs little to no extra time to understand and use the check code since ReDi only uses built-in programming language constructs. There will also not be a problem with the model and code being out of date since the code defines the model.

The same code analysis that ReDi perform in this thesis could also be applied to other languages since we have based all the principles described on standard programming language directives. We chose C# as a host language for this thesis since it has a rich type system, and it is easy to write an extension for the compiler. It is also a language used in many types of applications and used in the industry.

The main contribution of this thesis is to add a subtype-system based on formal-specification and symbolic execution. The subtype system can compensate for

language and runtime limitations as well as checking the code against the define restrictions. Improved type checking allows developers and system architects to easier write rules for the intended execution of a software package. We also tested ReDi against a C# open source project called CoreWiki created by Jeffrey T. Fritz [10], and several web service API projects for services offered by Innovation Norway, which was created by Bouvet.

The code for the thesis can be found at `https://bitbucket.org/Nicro950/nicroware.analzer.master/src/master/` [9].

1.1 Contributions and Outline

Summary of the contributions:

- Implement a Roslyn analyzer called ReDi for providing support for domain restrictions
- Developing principles for adding domain analysis for variables in programming languages
- Introduce the basis for a subtype-system which allows a domain restriction to bind to a variable
- Implement a richer type system for an existing language

Chapter 2 discuss some of the available technologies for improving the correctness of software. These include formal modeling languages, design by contract languages, and symbolic execution. We also discuss what technologies we have used in this thesis to perform analysis on code projects.

Chapter 3 looks at some examples which could throw errors at runtime and some solutions using ReDi for reporting the same errors at compile time.

Chapter 4 describes the overall design of the ReDi and the theory we used in the implementation of ReDi.

Chapter 5 discusses the different components of ReDi and how they interact with each other.

Chapter 6 reveals the results and discuss the usefulness of ReDi to find bugs and problems.

Chapter 7 is about differences to existing technologies and future work.

Chapter 8 concludes of the thesis.

Chapter 2

Background

In this chapter, we discuss the basic principles behind model checking and some technologies aimed at software correctness. We start by defining syntax and terminology that we use later on in this thesis. We then move on to discuss the following principles and technologies: symbolic execution, formal specification, formal modeling languages, and design by contract languages. At the end of the chapter, we discuss the chosen technology stack and some reasons for choosing C# and the .NET platform.

2.1 Terminology and Symbols

In this thesis, we use the following symbols and operations. We write a set as {`item1, item2, item3`} with curly braces. We write a range or interval as [a..b] where [and] are inclusive or with (a..b) where (and) is exclusive. We can also combine both as [1..5) which result in the set {1, 2, 3, 4}.

2.2 Domains and Restrictions

One of the most fundamental principles in this thesis is the domain of a variable, a restriction for a variable domain, and a set of restrictions. A set in this context is a mathematical set which is a list of unique items. The base domain of a variable is all the values a variable can have which fits inside its memory area. All variables then that have a base domain defined by their memory allocation. We also use the concept of a restricted domain, which is a subset of the variables original domain. Some examples of base domains are, for instance, a boolean variable that must have a value from the set of {true, false}. A 32-bit signed integer value, in contrast, has a much larger memory area, and hence a broader

domain and the domain can be specified as the range $[-2^{31}..2^{31}]$. A restriction of a 32-bit variable domain could be the range [0..1000) which contains all positive number with less than 4 digits.

The domain of a string is a little harder to restrict since a string is an arbitrary length array of characters. Each character in a string has a separate domain, which depends on the text encoding used for that character. The way we then restrict the domain of a string is by defining a set of legal lengths for the array, and the restrictions of each character domain. The easiest way to restrict a string is by using a regular expression or using a more advanced string parser.

A restriction is an invariant for a variable. An invariant is a rule which the code always should verify to be valid. We use invariants to define subtypes, which restricts the domain of variables. For instance, if we have an integer variable, we can define a subtype or invariant that requires that the variable is between 0 and 100 for the variable to satisfy the invariant. We can use an if statement to check if the variable satisfies the invariant, and if it does, we can annotate the variable with that specific subtype. A restriction is just an easy way of wrapping this concept into a clearly defined type, which ReDi uses throughout the code.

2.3 Methods for Improving Code Correctness

One method aimed at improving code correctness is using formal specification modeling languages such as TLA+ or Petri nets. Here a developer must define all the functionality of the system on an abstract level before the developer later implements it in code. These languages give a rich set of mathematical operations to define the different operations for the systems. However, these modeling languages are quite different from the implementation languages. This difference could lead to correctness problems when translating the model to the implementation language. Even if there could be a problem with the implementation, developers and researchers have shown that this reduces bugs, inconsistencies, and security problems [15, 18].

Another formal specification method is using design by contract languages such as Dafny and Eiffel. These languages make it possible to both represent code and model definitions in the same program code. Combining the model and code syntax integrates the model checker and compiler of the language more tightly but also makes it very hard to use the principles in other languages. Another problem with design by contract languages is that they, for the most part, binds the model checking at the function level, with pre and post-conditions. The pre and post-conditions are different rules which should apply to the input parameters and return value of a function. In contrast, much code today

binds the logic to a model instead, making a class or type invariant much more practical, which these languages also support.

Symbolic execution is a third option for improving code correctness. Symbolic execution calculates different execution paths through the software and tries to determine if any path results in an error or exception. The way symbolic execution accomplishes this is by replacing all variables that may change its value with symbols that get its domain restricted by conditional checks, such as if statements and while loops. Symbolic execution then executes the code in a virtual machine environment, which makes it easy to keep track of what the code does. If the execution in the virtual machine reaches a point where a symbol can cause an error, then the system backtracks to see if it can resolve a path of values to reach that specific point of error. Researchers have used a system such as KLEE for symbolic execution and have shown promising results when applied to the GNU Coreutils. A problem with symbolic execution is that it does not allow simply defining the behavior of a program. Developers can, on the other hand, use KLEE to find inconsistencies between different implementations in different code bases for the same program. Developers more often use symbolic execution for checking that the code does not crash because of problems with code.

There also exists external frameworks which try to implement the mentioned methods at runtime. A problem with external libraries is that it is usually too late to fix problems when the software is already in production. It will also require extra external libraries to build the code [17, 12].

2.4 Symbolic Execution

Symbolic execution is a software analysis technique which replaces fixed variable value with a symbol with the domain to the variable type [2]. For a 32-bit signed integer value, the symbol would have the domain of all integers in the range $[2^{31}..-2^{31}-1]$. Symbolic execution then executes the code in a virtual machine environment, which has these symbols instead of actual memory locations. The virtual machine tries to restrict the domain of this symbol when a conditional branching operation like an if, or a while loop. When the execution comes to an assertion, it tries to find a value for the variables that would lead to the assertion in the real software. If this path exists, then an adversary can use the same path and crash the software. If the virtual machine comes to a point where a buffer overflow could occur, then the VM tries to determine if the domain of the index symbol can exceed the bounce of the buffer.

The example code in Figure 2.1 show a very simplified execution through the virtual machine. First, the code reads an arbitrary integer from somewhere, with the domain of a 32-bit signed integer. Then on the third line, the code checks if a value is larger then zero, resulting in the symbol's domain changing.

At line number 9, the symbolic execution reaches a point where the program can crash and then tries to resolve the values leading to this point. The symbolic execution backtracks the value of `a` to the `ReadInt` and can determine that if `a = 1`, then this code crashes.

```
1  int a = ReadInt();
2  // symbol(a) = [(-2^31)..(2^32-1)]
3  if(a > 0) {
4      // symbol(a) = [1..(2^32-1)]
5      a = a - 10;
6      // symbol(a) = [-9..(2^32-11)]
7      if (a == -9) {
8          // this is part of domain and can happen
9          throw new Exception();
10         // Backtrack values
11         // a = -9
12         // a = a + 10 = 1
13         // If ReadInt returns 1, then crashes on throw
14     }
15 } else {
16     // symbol(a) = [(-2^31)..0]
17 }
```

Figure 2.1: Execution of a symbolic execution

Symbolic execution is a powerful tool to test if the software has any design flaws related to the programming language, but it cannot find business logic related errors. The compiler needs to know the intention of the code to capture the business logic error, which symbolic execution by itself does not provide.

2.4.1 KLEE

KLEE is a tool used for symbolic execution which uses restricted symbols to check if the code is valid [2]. KLEE has some similarities with ReDi, but there is one fundamental difference. KLEE is a tool created to verify existing programs and does a good job of doing so according to the KLEE paper [2]. The difference between KLEE and ReDi is that KLEE focuses on not changing the already defined code and tries to detect errors according to the C runtime. KLEE also can generate unit tests to achieve high test coverage. KLEE also has support for cross-validating two programs which should have the same functionality and detect the inconsistencies between the implementations.

Researchers have shown that symbolic execution is a reliable tool to automatically generating unit tests and checking that the code functions as it should[2]. KLEE is also usually getting a higher coverage score than ordinary developers written unit tests.

2.4.2 Hacker Tools

Security researchers use symbolic execution for finding bugs and logical flaws in software[5][1]. The software will, in these scenarios find different execution paths through the software, which could lead to different bugs. Hackers can then exploit the bugs and problems in several ways. One way could be for a code injection attack where a buffer overflow could be used to write code into the code memory block of the software. Denial of service attacks is another attack vector if a path through the software could lead to the software itself crashing. An adversary might exploit some of these paths to circumvent security enforcement and gain access to restricted areas or sensitive information.

2.5 Formal Specification

Formal specification is a technique for defining the intent of a model. These models can both be high-level models defining how the system function or lower level models defining how an individual component functions. Formal specification usually uses a mathematical language, or a language derived from it to define the rules.

There are several modeling languages available and classified as two groups. The first group is graphical languages such as Unified Modeling Language and Petri nets. These languages often have vertices and edges, such as a graph which describes the relationship between the different components of the models. For the text-based languages, there is TLA+ and Boogie.

2.5.1 TLA+

Temporal Logic of Action (TLA) is a formal specification language created by Leslie Lamport [18]. The TLC run the TLA+ model to calculate the states of the models and see if any statements violate the invariants. An invariant is a rule that the software should never violate through the running of a program. There also exists loop invariant in some languages which only restricts the running of the loop itself.

One of the main problems with common programming languages is that they are bad at describing non-determinism. That means that a programming language, in general, is bad at describing what something returns if the code cannot define the return value by a type. For instance, a Random function which returns a random floating-point number between 0 and 1 has no way of describing this behavior. The only way for anyone to know this behavior would be through the documentation, even if it is a central part of the function itself.

In TLA+ the engineer would define the restriction of a variable as an invariant. For instance, if a value should only contain the values between 0 and 100, then

the invariant would define the value as part of the set between 0 and 100. The TLA+ for the invariant is expressed in Figure 2.2.

```
1 TypeOK == somevalue \in 0..100
```

Figure 2.2: TLA+ code for defining the invariant for the domain of TypeOK

Here the variable `TypeOK` would be an invariant for the module which defines that `somevalue` should never contain a value outside the set 0..100. Defining invariants for variables helps with determining if a program invalidates its state and also helps the model checker to produce the steps that resulted in the invalidation of the invariant. The way the checker checks for violation in Figure 2.2 is by tracking the state of the `TypeOK` variable. If the variable state becomes violated, then the model is also violated. Finding violated invariants makes it easier to fix the current model and for creating a correct specification for the model.

A developer must then implement the defined model in a programming language to get the benefit of the model. Implementing the model could be a source of error if the developer does not implement the model correctly inside the code, which could lead to incorrect behavior, despite a correct TLA+ specification.

Writing TLA+ specifications is a complicated and time-consuming task. It gives benefits in the situations where the model could check a complex component. However, for other problems, it could be easier to have an ordinary unit test to check if the developer has implemented the component correctly.

Amazon Web Services is one of several companies which have found the use of the TLA+ model checking toolkit to improve the code quality [15]. They have now started to use it for checking parts of their services and has found inconsistencies in the code and fixed them with the help of TLA+.

One of the most significant benefits of using a TLA+ model is that a model checker if the state space is manageable, can determine if the definition violates any of the rules. If the model checker confirms that the model does not violate the definition, then the model checker proves that the current specification is correct.

2.6 Design by Contract

Design by contract is a software principle where the different modules of a program create contracts which the module expects all other code to conform to [14][6]. Design by contract languages, such as Eiffel and Dafny, enforces formal specification with the pre- and post-conditions, and invariant. For these languages, the contract is between a routine such as a function or procedure and

the routine invoker. The pre-condition is all the rules which the code must have verified when the routine starts, and it cannot run without these rules verified. The post-conditions are rules which the routine itself defines to uphold when the routine returns. The return conditions make it possible for the caller to know what to expect from that routine.

Invariants are a set of rules which the code should uphold for the context of the invariant. For instance, if there is a loop invariant, then these rules should be true in every step of that loop. There is also class invariant which applies to the fields of the class.

2.6.1 Eiffel

Eiffel is a programming language first released in 1985 and was the first language to incorporate design by contract [7]. Eiffel is an object-oriented programming language which incorporates the principles of pre-condition, post-condition, and invariant. Some types of invariants are the loop invariant and the class invariant [6][8].

In Eiffel, there are more strict rules for routines. A routine can in Eiffel either be a function which returns a value but do not change the state, or a procedure that does not return a value but changes the state. Eiffel calls this distinction for command query separation where a procedure is a command to change an object, and a function is a query to ask about the attributes of the object. A reason for command query separation is to know if there are side effects of a call. If it is a function, it should change no state, and a developer can call it safely. A procedure changes the state of the object.

Eiffel calls all the fields in an object for attributes instead of fields since a real-world object has different attributes related to itself.

2.6.2 Dafny

Dafny is a Microsoft Research language which implements Design by Contract [4]. Dafny is like Eiffel in that it implements pre-conditions, post-conditions, and invariants. Dafny is an imperative and sequential language with support for generic classes, dynamic allocations, inductive datatype, and built-in specification constructs [3].

The verifier that Dafny utilizes is power by Boogie and Z3, and the verification is directly integrated into the language itself. A directly integrated verifier results in that the developer handles the type errors produce by Dafny by changing the source code. Dafny also compiles against the dotnet infrastructure and to .NET Common Intermediate Language (CIL). The Dafny team develops the Dafny compiler in C#.

2.7 C# and .NET

We chose C# and the .NET platform for ReDi in this thesis, since the C# language has a well-defined type system, and it is easy to write plugins for the compiler. C# is also a mature language, open source, and have good tooling support.

Visual C# is a programming language created by Microsoft as part of the .NET family of languages in 2001[11]. The specification of C# has been open source from the beginning, and the C# compiler was open sourced in April 2014 under Microsoft Build 2014 developer conference. C# is a multi-paradigm programming language where some of its features are object-oriented, functional, generic, strongly typed, and imperative. Dotnet foundation owns C# and the C# compiler where Microsoft is the primary maintainer of the language.

2.7.1 C# Compilation

There are two steps for running C# code. The first step is using the Roslyn compiler to parse and compile C# code to Common Intermediate Language (CIL) code. CIL code is a low-level object-oriented intermediate assembly language which is quick to compile to machine code on the desired platform. The second part requires a Common Language Runtime (CLR) for that specific platform. The CLR loads the code into memory and starts a Just In Time (JIT) compilation to compile CIL to native machine code. The CLR then executes the machine code inside the CLR environment with accesses to the .NET Base Class Library (BCL).

2.7.2 The .NET Compiler Platform, Roslyn

The .NET compiler platform, also known as Roslyn, is the current official compiler for the C# and Visual Basic programming language. These compilers are self-hosted compilers, and Microsoft wrote them to get standard tooling for supporting analysis and compilation of the mentioned programming languages. Roslyn has two different parts, where the first is the compiler and the second is the analyzer part. The analyzer part plugs into the compiler pipeline to allow analyzing the code according to style guidelines and library rules. The analyzers should give better support for using libraries with giving help and code fixes to fix the code at compile time.

2.7.3 Roslyn Analyzer

A Roslyn analyzer is a plugin for the C# compiler which allows a more extensive checking of the code. The way the analyzer works is by getting a callback

in the different parts of the parsing process. There are two parts of the parsing processes where the first is converting the code into a syntax tree, and the other is converting the code into symbols. A symbol in the context of a compiler is extra type information related to classes, function, fields, methods, and properties.

An analyzer can walk through the syntax tree and then ask the compiler to get the symbols for specific parts. The symbols contain more information about the code used. For instance, a function symbol contains information about where the function is defined, and which input parameter return type the function has.

Figure 2.4 represents the syntax tree for the code in Figure 2.3. The syntax tree is recursive and contains various types of sub nodes which an analyzer must handle separately. The Figure 2.4 represents three different kinds of entries. The first entries are the blue entries, for instance, the `CompilationUnit` and `NamespaceDeclaration` which represents the syntax nodes. Syntax nodes are the main building block of the syntax tree and contain different high-level information such as classes, function, and statements. The next type of node is the green nodes, for instance, the `NamespaceKeyword` and `OpenBraceToken,` which are the `SyntaxTokens`. The `SyntaxTokens` comes from the parsing process of the compilation and represents the different parts of the syntax nodes. These parts can be the public keyword, the text representing the name of the function, or the open and close parenthesis.

Last part of the syntax tree is the red nodes which is the `Trail: EndOfLineTrivia` in Figure 2.4 which is the `SyntaxTrivia`. The syntax trivia is all other tokens which do not contain any information for the compilation. Some examples of syntax trivia are single and multi-line comments, spaces, and line breaks. The red entry in Figure 2.4 represents the end of line character at line 7 in Figure 2.3. The Roslyn compilers preserve the trivia in the syntax tree since it also has a function called code fixes.

```
1  namespace TestEnv
2  {
3      class SimpleCode
4      {
5          public void Run()
6          {
7              OtherCode.NeedsInRange(2);
8          }
9      }
10 }
```

Figure 2.3: Simple example code

Roslyn provides code fixes to give the developer automatic fixes for removing errors or problems found by an analyzer plugin. An example of a code fix is

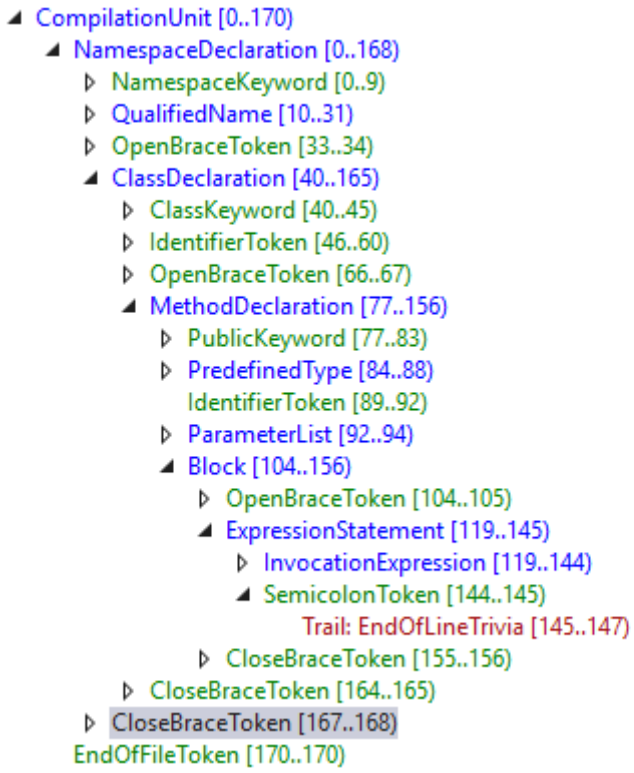


Figure 2.4: Syntax tree for code in Figure 2.3

that a team has decided that all class names should be all capital letters. Then the team could write a Roslyn analyzer which checks the class name for small letters and reports an error if it finds one. Then the code fix could take out the class name and convert it to uppercase and inserting it in the same place. We show the transformation for the code in Figure 2.3 which will output the code in Figure 2.5

```
1  namespace TestEnv
2  {
3      class SIMPLECODE
4      {
5          public void Run()
6          {
7              OtherCode.NeedsInRange(2);
8          }
9      }
10 }
```

Figure 2.5: Code after code transformation

In the example in Figure 2.3, it is important that the analyzer preserves all the spaces and comments. If the analyzer changes them, then it also changes the structure of the document, which could cause the code becoming harder to read. Figure 2.6 shows the same code as Figure 2.3 with almost no trivia. The compiler requires some trivia to differentiate between keywords and other tokens.

```
1  namespace TestEnv{class SimpleCode{public
2  void Run(){OtherCode.NeedsInRange(2);}}}
```

Figure 2.6: Code if an analyzer removed all trivia

Chapter 3

ReDi use Cases

In this chapter, we inspect some coding problems which could lead to faults. We present both examples which do not utilize ReDi for domain checking and a solution example which utilize ReDi. First, we are going to discuss some examples about simple array access and some solutions to safe array access with and without ReDi. Afterward, we discuss some problems and solutions related to null exceptions, and finally, we discuss some problems found the core wiki project.

The code can both seem slow and unreliable every time the runtime throws a runtime exception. A runtime exception could also lead to services becoming unavailable or the loss of data. A more common scenario is that developers discover the runtime exceptions while testing the software, something which increases the development time. Every time a software tester finds an exception, then a developer needs to use time on trying to replicate the error, figuring out why it happened, and come up with a solution of that scenario. For every runtime exceptions, a compiler or analyzer can handle on compile time results in less time being spent solving these exceptions. ReDi also adds more annotation, making it possible for developers to increase the descriptiveness of the code, which could result in the code being easier to read and easier to understand the intent of the code.

3.1 Accessing an Array Index

When accessing an array, it is important to know that the accessed index is inside the bounce of the array. For instance, in the example in Figure 3.1, we could get an exception if the index is out of bounce of the GlobalArray. Before this function can run, then the code needs to check that the input parameter index is a value inside the bounce of the array.

```
public static int GetElementFromArray(int index) {
    return GlobalArray[index];
}
```

Figure 3.1: Very simple problematic array access

The example in Figure 3.2, 3.3 and 3.4 shows some ways to handle the example in Figure 3.1. One way would be to throw an error or return an 'impossible' value as in Figure 3.2, but both are flawed methods since one requires the slow try-catch and the other requires extra non-intuitive checks. If the function returns a 0 value then, it is impossible to know if the value at that index was 0 or the index did not exist. The same problem would occur with the `int.MinValue`. It could also be a problem when the function returns the value, and no check was in place to check if it was an 'impossible' value.

Figure 3.3 returns a tuple where the last variable decides if the index exists or not. The problem here is that the code can use the first value without checking the second.

The last function in the example in Figure 3.4 is nullable value types. A Problem with nullable value types is that they could be null, resulting in an `InvalidOperationException` if accessed, as well as wrapping the value type in another struct, increasing memory consumption.

```
public static int GetElementFromArray(int index) {
    if (index >= 0 && index < GlobalArray.Length)
        return GlobalArray[index];

// Some alternatives if the index does not exists
    throw new NullReferenceException();     // Throws exception
    return 0;                               // return 0 value
    return int.MinValue;                    // return impossible value
}
```

Figure 3.2: Safe array access with returning impossible value

```
public static (int, bool) GetElementFromArray(int index) {
    if (index >= 0 && index < GlobalArray.Length)
        return (GlobalArray[index], true);
    return (0, false);
}
```

Figure 3.3: Safe array access with returning tuple (int, bool) value

Figure 3.5 utilizes ReDi for analysis of the code. ReDi adds the `SubType` and `CheckMethod` attributes for providing domain restrictions and the domain restrictor respectively. ReDi links the string inside the attributes together to en-

```
1 public static int? GetElementFromArray(int index) {
2     if (index >= 0 && index < GlobalArray.Length)
3         return GlobalArray[index];
4     return null;
5 }
```

Figure 3.4: Safe array access with returning nullable value

force that the check method `IsInRangeOfArray` is used on the variable before the `GetElementFromArray` is invoked. In C# there is no easy way to enforce that one or more methods are invoked before another method is invoked. The `SubType` attribute, `CheckMethod` attribute and helper library are further discussed in Chapter 4.

As shown in Figure 3.5, ReDi can detect and highlight that the code does not perform the necessary range check when accessing the array, which the code explicitly defined that it should do. The parameter on `GetElementFromArray` requires that the passed in argument has the `InRange` attribute, which is checked for in the if statement. In this example, ReDi can guarantee that inside the if statement, the code can never throw an exception as long as the developer implemented the `IsInRangeOfArray` function correctly.

The example in Figure 3.5 in contrast to the examples in Figure 3.2, 3.3 and 3.4 clearly defines the expectation of the function without adding overhead to the runtime execution. Furthermore, with the help of compiler optimizations such as inlining of methods, the example in Figure 3.5 can be reduced to the example in Figure 3.6 at compile time.

3.2 Nullable Object

Null pointers were first introduced in ALGOL W in 1965 by Tony Hoare [16]. He stated at the QCon conference in 2009 that this was a terrible idea and he believes that it probably has cost between 1/10 to 10 billion dollar. The problem with `null` objects is that they operate on the same principles as the example in Figure 3.2 where the function returns an 'impossible' value. If the code does not check for `null`, then it could throw a runtime exception and crash the program if the variable is `null`.

In the example in Figure 3.7, it could be problematic since `p` could be `null`, as well as `p.Name` could also be `null`. We need to check for `null` before we can use this code.

Figure 3.8 and 3.9 demonstrates some solutions to the example in Figure 3.7. Both examples do the same thing where the example in Figure 3.9 is a C# specific way of return `null` or continue the chain if it is not `null`. Both options rely on returning `null`, but the code does not specify nullability anywhere that

```
public static int GetElementFromArray([SubType("InRange")]int index) {
    return GlobalArray[index];
}

[CheckMethod("InRange")]
public static bool IsInRangeOfArray(int index) {
    return index >= 0 && index < GlobalArray.Length;
}

public void Main(){
    int index = -10;
    int s = GetElementFromArray(index); // missing the - InRange Attribute
    if (IsInRangeOfArray(index)){
        int number = GetElementFromArray(index);
    } else {
        Console.WriteLine("Number not in range of array");
    }
}
```

Figure 3.5: Code that ReDi throws error do to missing InRange subtype

```
public void Main(){
    int index = -10;
    int s = GlobalArray[index];
    if (index >= 0 && index < GlobalArray.Length){
        int number = GlobalArray[index];
    } else {
        Console.WriteLine("Number not in range of array");
    }
}
```

Figure 3.6: Inlined version of code represented in Figure 3.5

```
public class Person {
    public string Name { get; set; }
}

public string GetTrimmedName(Person p) {
    return p.Name.Trim();
}
```

Figure 3.7: Basic Person class with problematic null exception code

the type checker in the compiler can check. When a function checks for null and then return, it only moves the null error around and could lead to more processing before the code checks for `null` and return an error.

```
public string GetTrimmedName(Person p) {
    if (p == null || p.Name == null)
        return null;
    return p.Name.Trim();
}
```

Figure 3.8: Trimming a object which is nullable

```
public string GetTrimmedName(Person p) {
    return p?.Name?.Trim();
}
```

Figure 3.9: C# alternative to Figure 3.8

We demonstrate the use of nullability with ReDi in Figure 3.10 and Figure 3.11. ReDi provides the `Nullable` attribute from its helper library for developers to be able to annotate nullable objects. When ReDi is active in a project, then all objects are non `null` by default. The nullability and helper library are further discussed in Chapter 4.

ReDi threats `null` as a domain expansion resulting in a normal reference type cannot be `null` by default. Non-null by default means that there must be a specific declaration for a reference type to be `null`. From the example in Figure 3.10, `p` will report a `null` error since ReDi detects that `p` can be `null`. After the null check, the `p` value can no longer be `null` and ReDi does not report an error.

The big benefit of ReDi being able to detect variables that could be `null` is the prevention of null reference exception. Null reference exceptions increase the development time of the software since they are only discoverable on runtime. When an exception is only discoverable on runtime means that it either require manual testing or a unit test to check for nullability. By removing the possibility for a reference to be `null` then ReDi are basically freeing up the time required for checking and development of null errors. It also prevents a null error from being undetected in development and reaches the production environment. A null exception that happens in production could result in the software not responding or operating correctly. A fault in production could have critical consequences in the worst case scenario depending on the customer and use case.

Figure 3.11 represents another alternative to non-nullable references, by assigning that the properties and return value can be `null`. The difference here is that there is both a null check inside the function and a null check for the return value, doubling the number of required tests. In contrast to normal C# code, the example in Figure 3.11 explicitly specifies what can be `null` and what cannot.

```
1  public class Person {
2      public string Name { get; set; }
3  }
4
5  public string GetTrimmedName(Person p) {
6      return p.Name.Trim();
7  }
8
9  public void Main(){
10     Person p = null;
11     string s = GetTrimmedName(p);
12                          // p missing restriction -NonNullable
13     if (p != null) {
14          string s2 = GetTrimmedName(p);
15     } else {
16         Console.WriteLine("p is null")
17     }
18 }
```

Figure 3.10: Person class with ReDi support

The explicit specification makes it easy for ReDi to adjust the different domains to include null information, for ReDi to provide better nullability reporting. In the example in Figure 3.11, ReDi will report an error on the `s.Spilt(...)` statement. ReDi reports the error since the `GetTrimmedName` function explicitly states that it returns a value that can be `null` and ReDi cannot verify that there is some code in place that guarantees that `s` is not `null` when the code invokes the `Split` function. The code, as shown in the figure, results in a potential null error which ReDi does not allow.

3.3 Examples from CoreWiki

The CoreWiki project is one of the projects we used for testing ReDi against a code base. Jeffrey T. Fritz created the CoreWiki project as part of his twitch stream. The project is open source and hosted at GitHub [10]. The CoreWiki project is a Wiki software which Jeff wrote in .NET core, and which Jeff meant as a learning project for developers. The code snippet in Figure 3.12 represents an internal article query. At line number three, the code asks the database for an article with the given id. If the article exists, then the code return is, else `null` is returned. Afterward, at line 5 there is a check to see if `article` is `null` or not, if it is `null` then return `null` and if not then return the article converted to a domain object with the `toDomain` function.

By using ReDi, we were able to discern that this function could return `null`

```
1  public class Person {
2      [Nullable]
3      public string Name { get; set; }
4  }
5
6  [Nullable]
7  public string GetTrimmedName([Nullable]Person p) {
8      return p?.Name?.Trim();
9  }
10
11 public void Main() {
12     Person p = new Person();
13     string s = GetTrimmedName(p);
14     string[] parts = s.Split(' ');
15                      // s missing restriction -NonNullable
16     if (s != null) {
17         string[] parts2 = s.Split(' ');
18     }
19 }
```

Figure 3.11: Person class with nullable attributes

quickly, and we had to add the `Nullable` attribute on line 1. Since we added the `Nullable` attribute, we also got an error on line 12 reporting that `thisArticle` does not contain the `NonNullable` restriction. A null error could result in that the user of the application got an internal server error from the web server if the code gives `GetArticlesToCreate` an articleId which did not exist in the database. It is highly unlikely that the code can call this function with an `articleId` which does not exist from an external user accessible endpoint. What could happen is a developer trying to use this endpoint and give in an `articleId` which did not exist, and then have to debug and troubleshoot the application when it crashed. Fixing the error would use time which could be spent in maintenance of the code or adding new features.

Another example from the CoreWiki project is the removal of a null checks. Figure 3.13 show a method which either creates a new article or edits an existing article based on a slug received from user input. The code redirects to the already defined article if the code finds the slug in the database. A slug, in this case, is a URL friendly string of the topic for the article. The code in Figure 3.13 uses the function defined in Figure 3.14 for converting slug back to the original topic of the article.

In the example in Figure 3.14 we were able to add a `StringCheck.HasValue` subtype to the input parameter to guarantee that the input has a value. The `StringCheck.HasValue` subtypes requires that the input is first checked by the `IsHasValue` shown in Figure 3.15 before it is used. The `IsHasValue` in

```
1  [Nullable]
2  public async Task<Article> GetArticleById(int articleId) {
3      var article = await Context.Articles.AsNoTracking()
4                                 .FirstOrDefaultAsync(a => a.Id == articleId);
5      return article?.ToDomain();
6  }
7
8  public async Task<(string,IList<string>)> GetArticlesToCreate(int articleId) {
9      var articlesToCreate = new List<string>();
10     var thisArticle = await _repository.GetArticleById(articleId);
11
12     if (string.IsNullOrWhiteSpace(thisArticle.Content))
13     {                                // thisArticle missing restriction - NonNullable
14         return (thisArticle.Slug,articlesToCreate.Distinct().ToList());
15     }
16 }
```

Figure 3.12: Problematic code from the CoreWiki project

Figure 3.15 returns false if the given string is null, has length equal 0 or only contains white spaces. From the Figure 3.13 and Figure 3.14 we can see that the slug attribute is checked twice, first time at line 2 in Figure 3.13 and the second time at line 3 in Figure 3.14.

When we added the attribute to the parameter in Figure 3.14 as shown in Figure 3.16 we were also able to remove the string check that was redundant after the addition of the subtype to the parameter. Since we added the subtype attribute then we also needed to perform the check in the code depicted in Figure 3.17. In the code in Figure 3.17 we utilize scope promotion by returning inside the if statement to remove the `!HasValue` as a possible restriction from the slug variable. We then called `UrlHelpers.SlugToTopic` as normal with the slug variable. The difference between this and the original code is that it is easy for ReDi to verify that the code always checks the input variable before the code calls the `SlugToTopic`. The ability to always guarantee the correctness of the input variables also prevents `SlugToTopic` from returning a null value. It also prevents the code from doing the same check twice, once before the function call and one inside the function call.

Since we removed null and an empty string as possible values from the input parameter, we were also able to remove the unit tests as shown in Figure 3.18.

```
public async Task<IActionResult> OnGetAsync(string slug = "") {
    if (string.IsNullOrEmpty(slug)) {
        return Page();
    }

    var request = new GetArticleQuery(slug);
    var result = await _mediator.Send(request);
    if (result == null) {
        Article = new ArticleCreate {
            Topic = UrlHelpers.SlugToTopic(slug)
        };
    } else {
        return Redirect($"/{slug}/Edit");
    }
    return Page();
}
```

Figure 3.13: Original function which used the SlutToTopic described in Figure 3.14

```
public class UrlHelpers {
    public static string SlugToTopic(string slug) {
        if (string.IsNullOrEmpty(slug)) {
            return "";
        }

        var textInfo = new CultureInfo("en-US", false).TextInfo;
        var outValue = textInfo.ToTitleCase(slug);

        return outValue.Replace("-", " ");
    }
}
```

Figure 3.14: Original SlugToTopic before the removal of the redundant null check

```
public class StringCheck
{
    public const string HasValue = "HasValue";

    [CheckNotNull]
    [CheckMethod(HasValue)]
    public static bool IsHasValue(string s)
    {
        return !string.IsNullOrWhiteSpace(s);
    }
}
```

Figure 3.15: The IsHasValue C# check method

```
public class UrlHelpers {
    public static string SlugToTopic([SubType(StringCheck.HasValue)]string slug) {
        var textInfo = new CultureInfo("en-US", false).TextInfo;
        var outValue = textInfo.ToTitleCase(slug);

        return outValue.Replace("-", " ");
    }
}
```

Figure 3.16: Adding subtype and removing redundant null check

```
public async Task<IActionResult> OnGetAsync(string slug = "") {
    if (!StringCheck.IsHasValue(slug)) {
        return Page();
    }

    var request = new GetArticleQuery(slug);
    var result = await _mediator.Send(request);
    if (result == null) {
        Article = new ArticleCreate {
            Topic = UrlHelpers.SlugToTopic(slug)
        };
    } else {
        return Redirect($"/{slug}/Edit");
    }

    return Page();
}
```

Figure 3.17: Redundant null check

```
[Theory]
// [InlineData(null, "")]   // Removed checks since they are
// [InlineData("", "")]     // now guaranteed by the subtype
[InlineData("one-two", "One Two")]
[InlineData("home-page", "Home Page")]
[InlineData("onetwo", "Onetwo")]
[InlineData("one-two-three", "One Two Three")]
[InlineData("él-sofá", "Él Sofá")]
public void SlugShouldBeATopic(
    [SubType(StringValidator.HasValue)]string slug,
    string expected_topic)
{
    var actual_topic = UrlHelpers.SlugToTopic(slug);
    Assert.Equal(expected_topic, actual_topic);
}
```

Figure 3.18: Adding subtype and removing redundant null check

Chapter 4

Design

In this chapter, we discuss the overall design of the application and how the different parts of the application interact with each other. First, we describe the helper library and why we needed one. We then explain the code analysis with the first-pass and main-pass analysis of the code. After the different passes of the analysis, we describe different kinds of scopes we use in ReDi, as well as how to calculate the domains and the sets of currently applied restrictions.

ReDi tries to add a new subtype system which can restrict the domain of different variables. ReDi uses some principles from symbolic execution were ReDi analysis the code line by line to determine the valid domain of the variables. ReDi also lets the developer annotate the different variables and functions with the restrictions for ReDi to better understand the intent of the program.

4.1 Design Process

The C# programming language contains a significant amount of syntax, which limits the number of features that we can build into ReDi in the time available for this thesis. We used the CoreWiki project and 4 web API and service projects from Innovation Norway to guide the design process and identify common fault patterns. With these guidelines, we started an iterative development cycle to ensure that ReDi supports analysis of the most common code design patterns.

4.2 In Code Design (Helper Library)

ReDi needs a helper library to provide better type checking since C# does not contain functionality for defining virtual subtypes or alias for types. Changing

external types, like types defined in libraries or built into C#, is not possible either, without re-compiling the library, or the C# base class library. These restrictions result in two different infrastructures for adding type information to types. The first one is the helper library, which developers need for annotating the project code with attributes to support virtual subtypes. The second is a method for adding type information to compiled libraries and comes in the form of a `Nullable.txt` and `Passthrough.txt` files. These two files make it possible to add framework or library functions for ReDi to check for null or perform passthrough. We discuss these principles later in this chapter.

4.3 Code Analysis

We split ReDi into two different parts, the first-pass and the main-pass analyzer. C# code needs two-pass analysis since C# allows all members on the class level to have an arbitrary order, in contrast to C, which requires the developer to order all members. This result in ReDi needs to analyze all field, property, and method declarations before ReDi can analyze the content of those members.

4.4 First-Pass Analysis

In the first pass of the analysis, ReDi receives a syntax tree from the compiler and extracts all methods, fields, and properties from the classes and interfaces and stores them in a structure called `FileInfoCollection`. The `FileInfoCollection` holds information for each file that ReDi has analyzed, so when ReDi analyses a changed file, it does not need to re-analyze all the other files that have not changed before a developer, or a tool changes them. The `FileInfoCollection` structure also functions as a central interface for searching for class information in all the analyzed files.

The `FileInfoCollection` has the three different substructures `ClassInfoStore`, `MethodStore`, and `TypeStore`. The `ClassInfoStore` contains information about all the different classes and their members found in the project. The information contains all the member subtype information as well as if a class member can contain or return a null value. The `MethodStore` contains information about check methods annotated with the `[CheckMethod(string name)]` or the `[CheckNotNull]` attributes. Storing the method information in `MethodStore` allows for easy checking if an invocation tries to use a check method, or tries to use a conventional method. The `TypeStore` only contains information about the different types found in the `[SubType(string name)]` and the `[CheckMethod(string name)]` attributes.

4.5 Main-Pass

When the first-pass has analyzed all the syntax trees, then the main-pass analyzes the content of the methods and constructors found in the same syntax trees. ReDi starts by finding the code blocks syntax node, which makes up the method body, and then analyses the syntax nodes in the block. The primary function for handling node checks is the `HandleNode` function, whose primary responsibility is to check the type of the current syntax node. When ReDi has determined the `SyntaxNode` type, then ReDi executes one of several handler methods to handle that part of the syntax. We discussed the `SyntaxNode` in Section 2.7.3.

A handler method is a method for handling one type of syntax node types. For instance, the `InvocationExpressionSyntax` which is a function invocation has the `HandleInnovation` handler method. The `HandleInvocation` handler method checks that all the argument has the correct subtypes as well as no argument violating a non-nullable parameter. Another example is `HandleAssignmentExpression` which handles the `AssignmentExpressionSyntax` which makes sure that all restrictions are correctly transferer from the right-hand side to the left-hand side of an assignment statement. If the left-hand side contains any fixed restrictions, then this should report an error.

Since the C# compiler is recursive, then the handler methods must ask the `HandleNode` method again if there is any syntax that is not handled by the handler method. In Figure 4.2 we show the execution of the code in Figure 4.1. The `HandleNode` has been called for each layer of the boxes, demonstrating the recursiveness of the function.

The handler methods have two main tasks which they need to perform. The first is calculating the scope of the current syntax and check if all the code is legal, so it corresponds with the currently defined subtypes and nullability. The handler methods other functionality is to mutate the current scope and calculate the returned scope of the current syntax. Return scope means that if there is an invocation syntax, e.g., a method call, then the function should return the scope and restrictions of the return type of the function. The return scope, in this case, should also contain the information about if the returned value could be null or not.

If-calculations are also a part of the main-pass analysis, but they are described later in this chapter.

4.5.1 Scopes in General

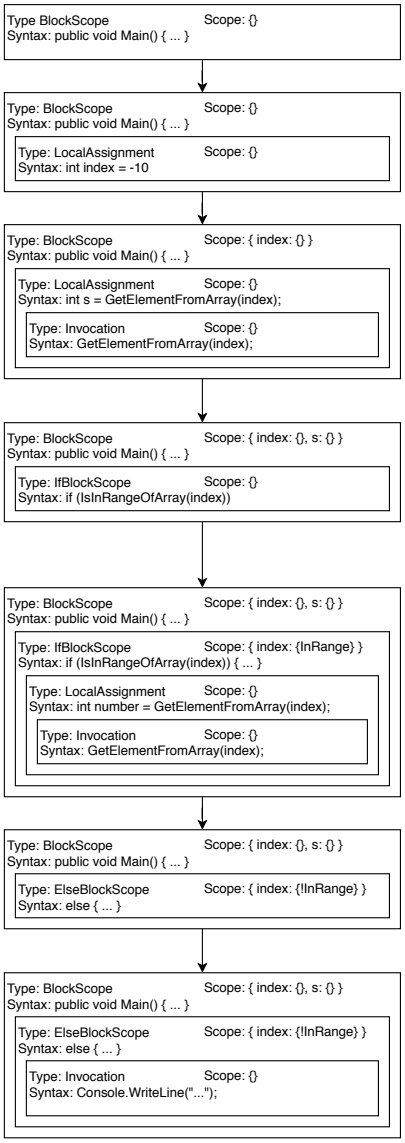
Scopes in the context of ReDi are recursive and only maintained while the analysis is running. The scopes are also mutable and ReDi mutates them throughout

```
public static int GetElementFromArray([SubType("InRange")]int index) {
    return GlobalArray[index];
}

[CheckMethod("InRange")]
public static bool IsInRangeOfArray(int index) {
    return index >= 0 && index < GlobalArray.Length;
}

public void Main() {
    int index = -10;
    int s = GetElementFromArray(index);
    if (IsInRangeOfArray(index)) {
        int number = GetElementFromArray(index);
    } else {
        Console.WriteLine("Number not in range of array");
    }
}
```

Figure 4.1: Simple code for exection

the analysis to keep track of the current state. Even if the scopes themselves are mutable, does not mean that all parts of the scope are. The scope definitions are recursive, and ReDi usually only mutates the first block scope, meaning that ReDi leaves the normal scopes as they are. Where the different scope types are described in Figure 4.3. There are many different scopes instances throughout the analysis, where each new piece of syntax has its mutable scope, which inherits from its parent scope. The scopes are implemented this way since it makes it easier to prevent code behavior from bleeding from one function over to another function. It also makes it easier for the outer scopes in the code, like a method scope to contain more information than an if-scope which is inside that method. The if scope should have access to the method scope to check restrictions of variables but should in most cases not change it.

Figure 4.2 demonstrates the recursiveness and mutability of the scopes, where each nested box is a new scope. The outermost scope is also changed based on what code is executed to keep track of the new state. ReDi also performs an argument check in the third box, where `GetElementFromArray` requires that the argument has the `InRange` subtype. None of the parent scopes can guarantee that the `index` variable has that restriction, which results in ReDi reporting an error.



Line: 10

The analyzer start analyzing the children of the function

Line: 11

-10 will also be analyzed and return an empty scope. Index will be added to parent scope with null scope

Line: 12

Here the second scope will request the return scope of the invocation. The invocation checks with the method store to see that **GetElementFromArray(index)** requires that its first parameter requires the **InRange** attribute, which ReDi cannot guaratee. This will throw an error saying that **index** missing the **InRange** attribute.

The function will also return an empty variable scope and assign it to s in the method scope.

Line: 13

Here the execution reaches an if statement which have the check IsInRangeOfArrayCheck(index). The if checks does not directly use the same HandleNode as the rest of the software and hence has its own handling.

The if handling will then check the method store to see if that method guarantees anything and add that to the ifscope. The if will add the InRange attribute to the if scope, and the inverted one to the else scope

Line: 14

When the code call the same function again, then ReDi tries to inspect the parent scopes to determin if index has the InRange attribute. This time, it is found on the if block level meaning that ReDi will not throw an error on this call. Since GetElementFromArray does not have any return scope then the ReDi will only add an empty variable scope for number to the if block.

This scope is now discarded since the execution of the if part is done.

Line: 15

Here ReDi starts analysing the else scope and adds the inverted restrictions to the else scope.

Line: 16

In line 16 we only write some output to the console window which does not require any restrictions.

Figure 4.2: An analysis of a function of the code in Figure 4.1

```
1  public void Main() {                                    // Block scope
2      int index = -10;                                    //   Normal scope
3      int s = GetElementFromArray(index);                 //   Normal scope
4      if (IsInRangeOfArray(index)) {                      //   Block scope
5          int number = GetElementFromArray(index);        //     Normal scope
6      } else {                                            //   Block scope
7          Console.WriteLine("...");                       //     Normal scope
8      }
9  }
```

Figure 4.3: Example which describes the location for the different scope levels

4.5.2 HandleNode and Return Scope

The current design of the implementation is that the `HandleNode` function is the primary resolver which decides which handler function should handle the current syntax. The handler method should return a return scope containing information about the resolved subtypes and nullability after the execution of the handler method. Figure 4.4 describes the flow of how the handlers interact with the `HandleNode` function. The figure describes the handle node execution of line 12 in Figure 4.1. In this execution, the `HandleIdentifierName` can determine that the input parameter has the `InRange` attribute and hence returns it to `HandleNode`. `HandleNode` then returns the variable scope with the `InRange` subtype to the `HandleInvocation` method, which verifies that the argument matches the requirement of the parameter. The `HandleInvocation` does not have any return type information and returns an empty set to the `HandleNode` which in turn returns it to `HandleLocalDeclarationStatement`. The `HandleLocalDeclarationStatement` received an empty scope from the `HandleNode` method and assigns the empty scope to the variable `number` as part of the if-scope.

The `HandleNode` method also tries to explore child nodes if there is currently no handler method available. We chose this architecture since it provides a central resolver for all C# syntax. If a handler method needs to resolve a scope for a syntax, then it only needs to call the `HandleNode` function. Another benefit with this approach is that ReDi analyses all the syntax nodes on the sub syntax even if there is no handler method for that syntax to providing more diagnostic.

4.5.3 Nullability in C

Nullability is a central component for providing better information about the code in conjunction with the subtype system. The problem with nullability in C# is that all class type variable can be null by default. Thus, two different approaches are possible, one which is nullable by default and one which is non-

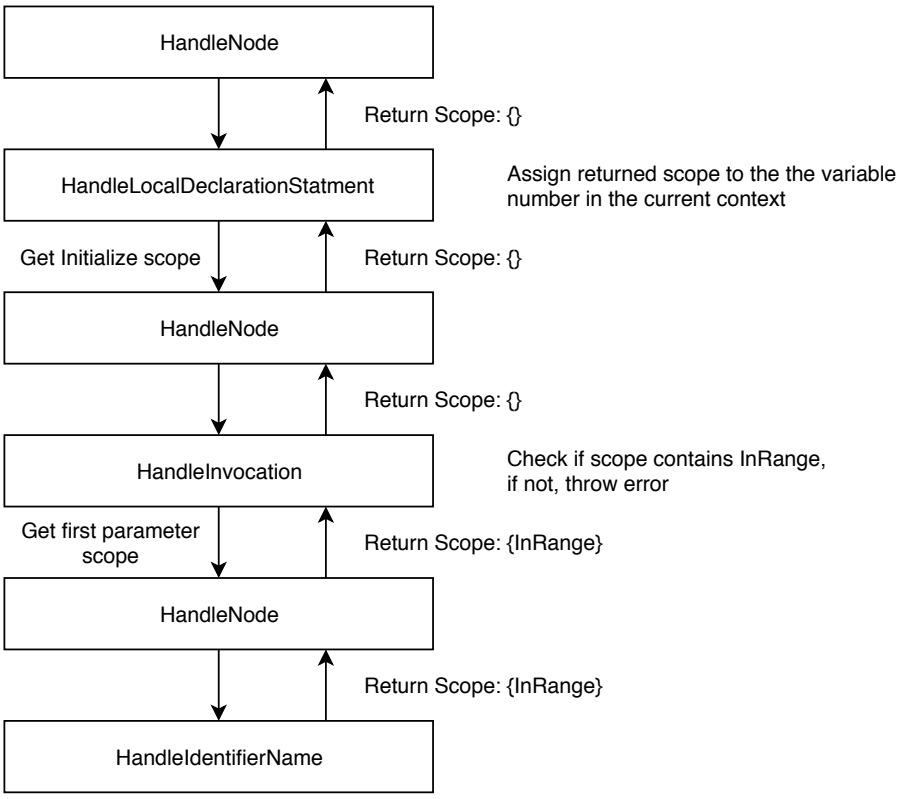


Figure 4.4: The inner workings of the handle node function

nullable by default. Non-nullable by default is both chosen in TypeScript and the next version of C#, which is C# version 8. Both C# and TypeScript refers to non-nullable by default as non-nullable reference types. The benefit of using non-nullable by default is that most classes are usually not null and if null by default should be used, then more syntax and code is necessary to annotate everything as not null. Choosing non-null by default also makes it fit nicely with nullable value types which already exists in C#. Value types such as int, long, float, and struct cannot be null since they are stack based. C# has built-in support for making value types nullable with the `?` operator behind the type name such as `int?` or `bool?`.

Even if C# version 8 implements non-nullable reference types, it does not seem to implement enhancing external libraries, which makes code from external libraries more dangerous to use.

C# does also have a couple of different approaches for dealing with nullability, which also ReDi must handle. The null coalescing operator `??` is one implementation which checks if the left-hand side is null, and if true, returns the right side. Figure 4.5 describes the functionality of the `??` operator. Scope checking for the null coalescing operator would result in an intersection of the two possible scopes being return. If the right-hand side cannot be null, then the result can be guaranteed never to be null.

```
1 function Test(left, right){
2     if (left != null) return left;
3     return right;
4 }
```

Figure 4.5: How the null coalescing operator functions

Another operator is the Null conditional member access operator `?.` which only continues down the member access path if the left-hand expression results in not null. In the code, this operator could result in the statement being nullable if anything on the left-hand side could result in null. Here ReDi resolves the expression and ensure that the result of the operation results in a nullable type.

4.5.4 Passthrough

Another feature which ReDi needed to support was the passthrough functionality. The passthrough feature is needed to support the factory pattern, where a type method returns the `this` instance. `Task<T>` is one class which have methods implementing this pattern, and developers often use `Task` in asynchronous workload. The passthrough attribute makes it so that ReDi returns the context information of the `this` object as the return-scope.

4.5.5 Only Analyzing Changed Files

In the CoreWiki project, there are over 300 files, where a developer only changes one file at a time. It would waste many resources if ReDi should analyze all the files each time, which would result in poor performance. To circumvent this, ReDi only analyzes all the files the first time, and then only the files changed afterward. All the files need to be analyzed the first round since most of the logic is defined in other files and would in other cases not be available before the editor opened that file.

4.5.6 Not Necessary Checking

C# has built-in type-checker which can verify the basic C# syntax and the defined types. The type-checker simplifies the process of creating subtypes since they are inherently bound to the C# type using the check function. If there is a check for if a string contains any characters and then get the subtype `HasValue`, then the check and subtype would be bound to a string. ReDi does not need to keep track of the C# type the restriction is bound to since the code fundamentally links them in the check method.

If someone should implement the subtype system in another language without a strict type-implementation, then the basic type-information would need to be checked as well. Without having a strict type system could result in a check method accepting the incorrect input type and, either due to a bug or other error, give the subtype to the incorrect type. Binding the subtype to the wrong type could lead to more severe faults later in the execution and would defeat the purpose of having the extra check there in the first place. In languages without a strict compiler aware type system, the subtype would both need to restrict the domain of the variable down to a specific type implementation and to the desired domain of that variable.

4.5.7 Query and Response or Atomic Operations

When using the subtype system, there are two different methods for implementing queries against a data structure. The first method would be first to ask if a given value exists and then request it in a next statement or ask for it and get an optional value back. In C#, developers often use the first pattern for Dictionary access, which is a map type, where the developer first queries if a key exists, and then ask for the value at that key. This pattern works fine for single-threaded application but could lead to race conditions if it is multithreaded. The subtype system supports this scenario with linked restrictions and the possibility to link a variable restriction to the structure it tries to access. The problem transpires if another thread deletes the key before the next statement executes resulting in that the value request would fail because the key no longer exists.

The other implementation of this would be to ask for the value and either get the value back or get an empty value, such as null, back. A problem with this pattern is that there always needs to be a check after this to see if the value exists or not. Instead of the operation returning null, it could also return an option type, which would wrap the value in another structure with information about if the key exists. The option type would still need to be checked afterward but would never be null. The options pattern could be supported by linked restrictions, which links the value exist check if to the value access, requiring the check before retrieving the value. ReDi also supports the dictionary returning null by marking the retrieved variable nullable. ReDi then enforces that the developer checks the value for null before the developer tries to access it. If the code does not check the value then ReDi reports an error to the developer specifying that the variable could be null.

4.6 Logic Scope

All the code in the syntax tree is recursively defined and hence needs a recursive checking. We separated the scope-implementation between block-nodes and normal-nodes. Figure 4.3 demonstrates where these scopes are where block-nodes have a block-scope and normal-nodes have a normal-scope. Block nodes are the nodes that contain all the variable restriction information, while normal nodes are for the different levels of the syntax tree. As can be seen in Figure 4.6 there are many different levels to the syntax tree. Nodes which would be considered block nodes in the figure is the `ClassDeclaration` and `Block` syntax nodes. C# also uses these two node types for the regular variable scoping rules in the programming language.

4.7 If Condition Scopes

The domain of variables is changed every time an if-statement check them. This is the basis for ReDi which exploits this principle to perform an analysis of the code and provide better type restrictions. The IF-condition can be a series of chained together commands with the help of logical `AND` and `OR` operators. Both the `AND` and `OR` operators mutate the current IF-condition scope. The available operations in an if-scope calculation are the negate operator `!`, the Boolean `AND` `&&`, and the Boolean `OR` || operators.

4.7.1 AND and OR Scope

Different rules apply for the grouping of different variables. We call these scopes for `AND` and `OR` scopes. An `AND` scope is when checks are grouped inside an if

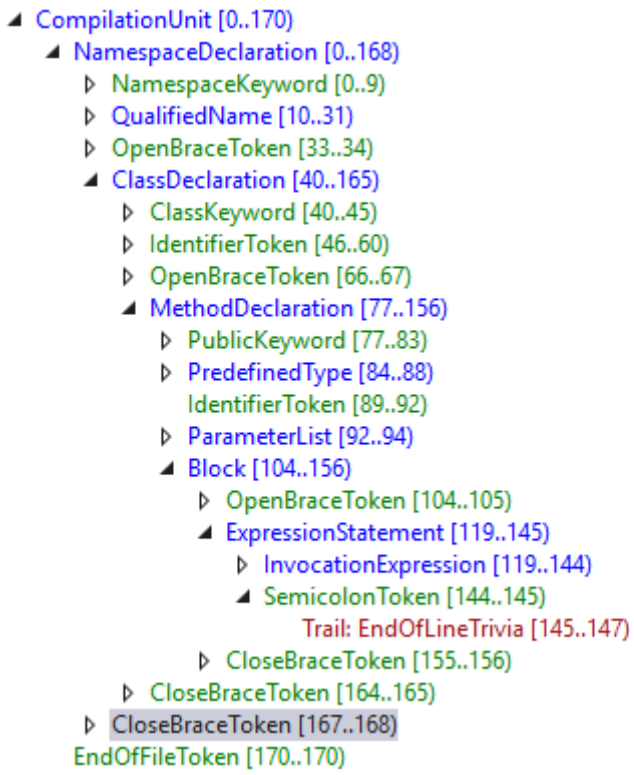


Figure 4.6: Syntax tree for a code snippet

condition with an `AND` operator. The `OR` scope is similar but is linked with an `OR` operator instead. The `AND`-scopes enforce merging the restrictions of the two scopes. `OR`-scopes take the intersection of the two scopes instead. What the if-calculation also do is taking the inverted scopes for the else clause. This means that the `AND` operator intersects the inverted restrictions for the else scope and `OR` scopes merges the inverted restrictions. The truth tables for AND and OR operations are described in Table 4.1 and Table 4.2 respectively.

Table 4.1: Truth table for AND operator

A	B	O
0	0	0
1	0	0
0	1	0
1	1	1

Table 4.2: Truth table for OR operator

A	B	O
0	0	0
1	0	1
0	1	1
1	1	1

The tables 4.1 and 4.2 shows that both tables only have one unique output. The sequence 1 AND 1 = 1 and 0 OR 0 = 0 is the only sequences that we can predict from the outputs. If we have an AND operation which produces 0, then there are three different cases which produce that output, and it is not possible to determine the input scopes. A path can also visualize the calculations through a graph. The Figure 4.7 represents the two states of `A` and `B` on the first line. These states can be combined to produce the combinations 00, 01, 10, and 11. When we apply the `AND` and `OR` operators to these combinations, we get the results 0 and 1. In the `AND` calculation, we only have one path leading to the 1 resulting in that we can predict the input of that calculation. We can predict the input of the NOT operator any scenario.

4.7.2 Merging of Scopes

If the variable `A` has two different check functions, for instance, `Email` and `Name`, and both can guarantee that it has a value. The check functions would then return the set {`IsValid, Name`} and {`IsValid, Email`}. If a developer checks this with an `OR` expression, then no matter which of them is the true one, we

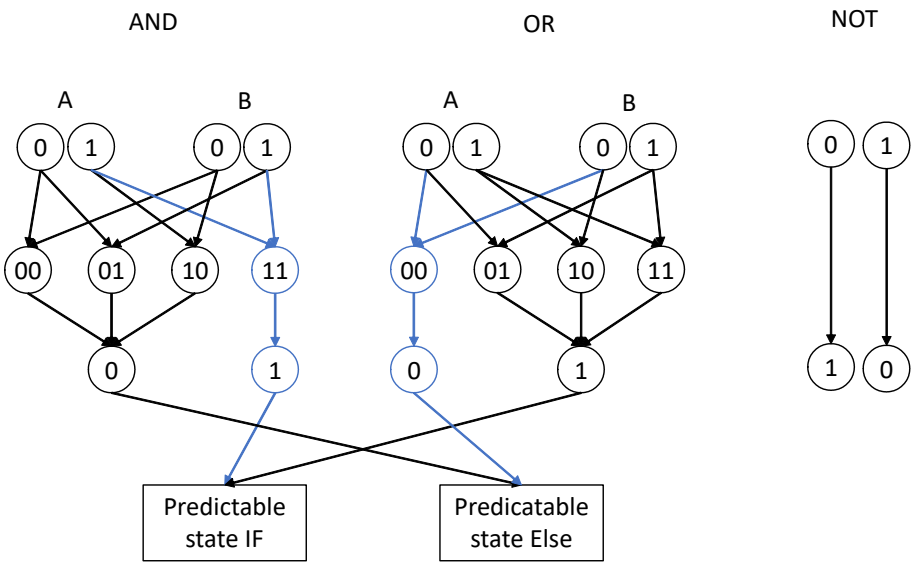


Figure 4.7: AND and OR path calculation

can guarantee that the variable `A` has the restriction `IsValid`. If a developer checks this with an `AND` expression, then we can guarantee both of them. For the else part, on the other hand, the OR expression can guarantee that none of the values exists, and merges the two sets {`!IsValid, !Name`} and {`!IsValid, !Email`} together to the set {`!IsValid, !Name, !Email`}. The `AND` expression functions similar but that can only guarantee the set {`!IsValid`}, since if the check failed, we do not know if `Name`, `Email` or both are false.

4.7.3 NOT Operator

The `NOT` operator does function a little differently than the `OR` and `AND` operators. The `NOT` operator switches the IF and ELSE scope with each other. If a developer uses the NOT operator for only a part of the expression, then only that part of the IF and ELSE scopes are switched.

4.7.4 Else Scope Promotion

If the IF statement returns or throws an exception, then the else scope could be promoted up one level since the state would never be recombined with the if state at the end of the if. Scope promotion allows for code patterns such as checking if an input parameter is null and return null at the start of the method and have the state set for the rest of the method.

4.7.5 Execution of an If-Scope Calculation

```
1 string s = ReadInput();
2 string s2 = ReadInput();
3 if (!(IsName(s) && IsName(s2)) || !(IsName(s) && IsEmail(s2)))
4     throw new Exception("Can not handle input");
```

Figure 4.8: Simple C# code snippet for describing if calculations

Figure 4.9 gives an example if-scope calculation for the code show in Figure 4.8

Since the example in Figure 4.9 throws inside the if, then an else scope promotion is performed and ReDi can guarantee that the scope for `s` is {`Name, IsValid`} and `s2` is {`IsValid`}.

```
Calculated scopes for "IsName(s) && IsName(s2)"
If scope
    s  => {Name, IsValid}
    s2 => {Name, IsValid}
Else scope
    s  => {!Name, !IsValid}
    s2 => {!Name, !IsValid}

Calculated scopes for "!(IsName(s) && IsName(s2))"
If scope
    s  => {!Name, !IsValid}
    s2 => {!Name, !IsValid}
Else scope
    s  => {Name, IsValid}
    s2 => {Name, IsValid}

Calculated scopes for "IsName(s) && IsEmail(s2)"
If scope
    s  => {Name, IsValid}
    s2 => {Email, IsValid}
Else scope
    s  => {!Name, !IsValid}
    s2 => {!Email, !IsValid}

Calculated scopes for "!(IsName(s) && IsEmail(s2))"
If scope
    s  => {!Name, !IsValid}
    s2 => {!Email, !IsValid}
Else scope
    s  => {Name, IsValid}
    s2 => {Email, IsValid}

Calculated scopes for "!(IsName(s) && IsName(s2)) || !(IsName(s) && IsEmail(s2))"
If scope
    s  => {!Name, !IsValid} ∩ {!Name, !IsValid} = {!Name, !IsValid}
    s2 => {!Email, !IsValid} ∩ {!Name, !IsValid} = {!IsValid}
Else scope
    s  => {Name, IsValid} ∩ {Name, IsValid} = {Name, IsValid}
    s2 => {Email, IsValid} ∩ {Name, IsValid} = {IsValid}
```

Figure 4.9: If scope explanation of Figure 4.8

Chapter 5

Implementation

In this chapter, we discuss how the implementation of the code works and what it does, also how some of the key concepts work in the code and why they work. First, we discuss the helper library and its features. We then discuss the restrictions system, and the different scopes, such as `VariableScopes` and `ContextScope`. After the different scopes, we discuss the principles behind nullability and the main flow of scope information through the `HandleNode` function

We implemented the analyzer tool, ReDi, as a Roslyn Analyzer which is part of the .NET compiler platform. ReDi works on the C# syntax, but the principles behind it can be adapted to other languages as well. ReDi provides real-time feedback directly in Visual Studio, which makes it very easy to use. Beneath in Figure 5.1 is a screenshot of how ReDi integrates into Visual Studio to provide feedback at compile time.

5.1 Helper Library

A helper library was needed, as described in the design chapter. This library defines some Attributes and Functions for helping with the use of the subtype system.

5.1.1 Attributes

The helper library adds the following four attributes:

- CheckMethodAttribute
  - Used to define a check method which can guarantee that a subtype contains the correct domain if it returns true.

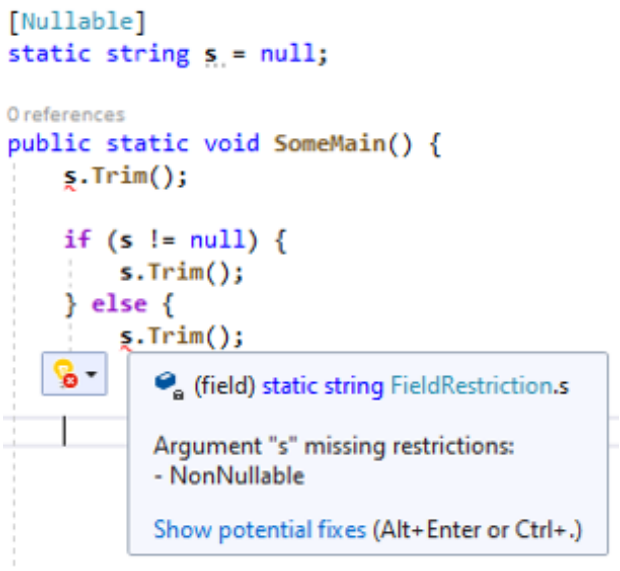


Figure 5.1: Vs code snippet

  - The Check Method is used in conjunction with an if statement to enforce that the SubType is correct inside the if context.
- SubTypeAttribute
  - Marks a member to require a specific SubType.
- NullableAttribute
  - Marks a member so that it can be null.
- CheckNotNullAttribute
  - A function which can guarantee that something is not null if it returns true

5.1.2 Helper Methods

`Scope.Check` was added since C# or Visual Studio does not have built-in support for inspecting a member's virtual types provided by ReDi, since this information is only available for ReDi. To circumvent this issue, `Scope.Check` takes in an object value and gives an information box on the current restrictions applied to the variable. Many of the unit tests for ReDi also uses `Scope.Check` to verify that a subtype has the correct subtypes in addition to the error messages that ReDi returns.

5.2 Code Implementations

Here the implementation of the different components of ReDi is discussed. Also, the interaction between the different components is listed. We start by discussing the central concept of a restriction. We then discuss the concepts behind `VariableScope`, `ContextScope` and `ContextInfo`.

5.2.1 Restriction

The restriction struct is one of the most fundamental types in ReDi, which contains information about a single restriction. A restriction contains three fields, which is `Name`, `Inverted`, and `LinkedTo`. The Name fields contain the name of the restriction and have to correspond with a check method. The Inverted field represents if the restriction is inverted, or that a variable is guaranteed not to contain that restriction. For instance, if a variable A should be in the range of an array, then it would have a domain like `a >= 0` and `a < length(array)`. If that check does not result in true, then ReDi can state that it can only contain a value outside that domain `a < 0` or `a >= length(array)`. ReDi uses this in if-scope calculations and gives the normal version to the IF-scope, while ReDi gives the inverted to the else scope as discussed in Chapter 4.

The `LinkedTo` field represents if it is a linked restriction. Linked restriction means that the variables restriction links to another variable. For since like in the array example, the `a` variable would be linked to the array it is checked against, but it is not necessary inside the range of every array in an application. Another example of a linked restriction is a key in a Map. If a map variable `map` contains a key `B`, then the `B` has the restriction `AKeyOf(map)`.

The Restrictions also have a standardized string format, making them easy to serialize and deserialize. The format is `[!]Name[(LinkedTo)]` where the parts in `[]` is optional. The negate symbol from many programming languages inspired the first part, which represents the inverted field in the restriction. The second part is the name of the restriction, and the last part in parentheses is a variable that is linked by the restriction.

`Restriction` is a struct in C# instead of a class, because they operate by pass-by-value instead of pass-by-reference. Pass-by-value makes the restrictions more immutable and harder to change accidentally. The other alternative would be to use classes and implement a copy function on the restriction. A problem would be if we did not call the copy function, which could result in there being two references to the same object, which could lead to problems with restriction mutations.

5.2.2 Linked Restrictions

The restrictions in ReDi can both be unlinked and linked. When it is unlinked, developers can use it in all places the code requires that specific restriction. If the restriction is linked, then it can only be used in conjunction with the linked variable. To be able to link a specific variable to another variable, a check function with two input parameters is required. The first parameter must be the variable itself that should have its domain restricted. The second parameter is the linking variable.

An example of using linked restrictions is the key of a dictionary. The C# language requires that the developer checks that the key exists before the code can access the key. To accomplish this, an if statement with a key check must come before the dictionary access itself. ReDi supports this checking with linked restriction. The way ReDi does this is by giving the key as the first parameter and the dictionary as the second parameter to the key check-method. Then it is up to the check function to check that the key is a part of that dictionary and returned true if it is. Then ReDi can guarantee that within the scope of the if statement where the code performed the check, the key must be a part of the dictionary. The problem with this approach is that it's not an atomic operation. Not an atomic operation means that the check pattern only works if it is a single threaded application or only one thread have access to the dictionary at the time. Else locks must be used to be able to lock the dictionary before the access to enforce the code follows proper concurrency patents.

Another use case for linked restriction is the ability to link a value to a range. For instance, in the C language, there is a problem with the ability to access indices of an array which is out of bounce. Indices that are out of bounce can lead to fatal problems where the application tries to read memory that it is not supposed to read. The C# language handles this a little bit differently and throws an index out of range exception if the code tries to access an index which is not in the range of the array. Even if this is a safer approach than just allowing the program to access whatever is at that memory address, it still crashes the program. The index check is also always performed at the runtime to verify that the index is inside the range of the array, which could lead to performance penalties.

Another approach could be to check that the index is part of the array bounce before the index is accessed, just like the dictionary example. Checking indices could use a similar approach as the dictionary example by using an additional check function. This function would take in the index as the first parameter and the array as the second. The function then performs a check to see if the index is in bounce. In a future version of ReDi, it would be possible to calculate the available domain of a variable before the code tries to access an array. Domain-calculations could result in the check being removed entirely making the application even faster.

5.2.3 VariableScopes

A `VariableScope` is two different collections of `Restrictions` for two different scenarios. It also contains nullable information as well as an `ObjectScope` for sub-variables. ReDi sorts restrictions into two different categories, the mutable restrictions, and the fixed, immutable restrictions. When the code performs an IF-check on a variable through a check method, ReDi adds a mutable restriction to the `Restrictions` `HashSet` of the `VariableScope`. If the variable is assigned a new value, then the restrictions are overridden with the scope of the assigned value.

In contrast, assignments cannot override fixed restrictions since those restrictions belong to the class level. These restrictions outlive the scope of a single function, which means that it would not be possible to keep track of the restriction mutations. ReDi forgets all calculated scopes and mutable restrictions from within the function when it returns, since it would result in a state explosion if the state should persist through these boundaries. Deleting the scopes also results in that a developer cannot change the domain information which belongs to class level.

To assign a value to a variable with a fixed restriction, then it needs to be first checked with the check functions to verify that the new variable has the correct restrictions. The required restrictions from the variable could both be fixed or normal if they are verified and not violated. The nullable field represents if the variable could be null or if it cannot be null. ReDi only tracks of a variable can or cannot be null and does not track if the variable is null, or can be null and are not, since this would require more processing.

5.2.4 ContextScope

A context scope is the collected variable information about a given scope. A ContextScope instance contains a dictionary or map with variable scopes to represents all the different variables present in the given code scope that have subtype information. The context scope also contains functions for merging scopes with intersections or union. The context scope instance is also the primary interface for manipulating the information stored in VariableScopes within the ContextScope.

5.2.5 ContextInfo

The ContextInfo class is a recursive implementation of a wrapper around the context scope. The recursive implementation means that it also contains a reference to another ContextInfo, which is its parent, like how folders in a file system work. ContextInfo needs to be recursive to be able to account for code scope calculation in C#. For instance, a method would have access to all the

different fields and properties declared at class level, but not the other way around. This means that an inner scope should have access to the information stored in an outer scope and that the inner scope information should not bleed out at any given time. This structure is also inspired by how prototype-based languages such as JavaScript and Lua creates objects and inheritance. The prototype-based languages do not have classes, but a prototype object which can contain more information. If the current instance of an object does not have a given field or method implementation, then the fields of the prototype are recursively inspected to check if it contains that implementation. If the prototype has the implementation or one of the prototype's prototypes has it, then it is invoked or accessed.

A similar process happens when ReDi requests a variable scope from a `ContextInfo`. The ContextInfo first asks its parent for calculating the scope and then merge in its information afterward. Merging scopes make it simple for child `ContextInfos` to provide more information on top of what the parent already knows.

5.2.6 Nullability Implementation

It was not possible to add features to the C# compiler since live analyzers such as ReDi can only analyze the code. Changes to the compiler would require recompiling the compiler and using the new compiler to provide analysis. It would also require that the C# interactions in Visual Studio were modified to support the new syntax, which would be outside the scope of the thesis.

The current implementation of ReDi makes use of the same principles as C# and TypeScript by providing non-nullable references by default. This behavior requires some additional checking to make sure that a nullable value is not able to be assigned to something that cannot be null. An example of this is when a method tries to return null, but the code does not mark the method with the nullable flag. ReDi then reports an error on the return statement saying that the method is missing the non-nullable attribute.

One thing about implementing it as non-nullable by default is that it is the opposite of how the restriction based subtype system functions. We based the already implemented subtype system on only reducing the available domain of the variable instead of expanding it. When, in contrast, making a variable nullable means that we increase the scope with an extra value instead of decreasing it. Since there is no other way of increasing the domain for a variable except nullability means that it is a special case. Domain expansion also resulted in that the implementation became a special case with the introduction of the nullable field in the variable scope.

5.2.7 Strict Nullability vs. Floating Nullability

The same problems arise with nullability as with subtypes. If the code defines a variable at class level, then it should have a strict nullability which cannot change. If the code defines a variable at a function level instead, then the variable could change nullability inside that function and ReDi would be able to track it. ReDi changes the nullability for a variable defined at function level if it is assigned null or something that can be null, making variables at function level have a floating nullability.

5.2.8 Null Scopes are Inverse of Normal Scopes

Null scopes are a domain expansion and hence works differently than the normal domain restriction subtypes. If a function returns a variable with a subtype one place but without it later in the function, then we cannot annotate the function with the subtype. On the other hand, if a function returns a nullable value once, then the entire function must be marked with the nullable attribute.

5.3 First-Pass of the Analysis

The first thing the analyze does when it receives the syntax tree is to locate all the class and interface declarations. Afterward, it is trying to find all the members of the interfaces and classes and do further analysis on them. In this case, both fields and properties are handled in the same way since developers use them in the same manner. Functions are analyzed a little bit differently since it has both input and output variables. First, ReDi checks if the function is a check method or has a subtype. Afterward, the parameters of the function are checked to see if some of them contain a subtype or nullability attributes. This information is then recorded in class information structure and store within the file information superstructure. ReDi also stores the return type information in the same structure. This information is quick to look up when the rest of the functions are analyzed. ReDi does not analyze the content of the methods in the first-pass since much information is still missing from the file information structure.

5.4 Main-Part Analysis

When the first-pass analysis is complete, the main analysis can start to analyze the function body. Is accomplished this by calling the `HandleNode` function.

5.4.1 HandleNode

We split the `HandleNode` function into 2 different parts. The first part is the `HandleNode` function itself, and the second is the `HandleNodeInner` function. The `HandleNode's` primary function is to execute the `HandleNodeInner` function with the input parameters. The `HandleNodeInner` function is a switch statement which takes in all the different C# syntax classes and calls the correct handler method. Then it is the job of the `HandleNodeInner` function to retrieve the return-scope of that handler function and return it to the `HandleNode` function. Other methods should not call the `HandleNodeInner` function. The reason for this is that `HandleNode` function has a secondary purpose. `HandleNodeInner` function or one of its handler methods could return a state that specifies that it should automatically explore the child nodes. The `HandleNode` function retrieves these results and calls the `ExploreChildNodes` function if a handler method has set the flag.

If the `HandleNodeInner` function does not have a handler for the given syntax, it returns the `HandleNodeState` with the `ExploreChildren` flag set to true. We implemented it in this manner to have a higher chance of providing more analysis. If ReDi comes upon syntax that it does have a handler for then it could still be that the child node is recognizable. Then we still want to try and perform analysis on the child node to get the best experience with ReDi. In some cases, we do want to ignore the child nodes, and this can be performed by just returning an empty `HandleNodeState` structure. The reason to ignore some `SyntaxNodes` is that they have already been analyzed and cannot provide further information. One instance of this is the InterfaceDeclarationSyntax, which ReDi analyzed in the first-pass of the analysis. If we allowed the analysis to analyze the same syntax multiple times, it would also become a problem with ReDi report the same errors multiple times.

5.4.2 Wrapped scopes

Wrapped scopes is another thing that was necessary to support certain coding patterns. The most prominent use case is the C# task system. When a function is async and returns a task, then we are not interested in giving type information about the task variable itself but the generic type provided by task. We provide this functionality by detecting that a function returns a task and assigns the type information in a wrapped scope instead. When the analyzer then reaches an await operator, we unpack the wrapped scope and return the inner scope instead.

5.5 If Scope Calculations

In Chapter 4 we discussed how the if scope calculation worked and here we are going to discuss some of the more implementation specific parts of the calculation. After the `HandleNode` invokes the handling of the if-statement and the handler has been given control, it first tries to parse the condition of the variables. The condition parsing is a recursive operation by starting with the first `SyntaxNode` reached. The if-statement handler creates two different scopes for determining the scopes, one for the if context and one for the else context.

The if handler takes the condition and passes it to a `TraverseIfCondition` function which recursively traverses the if condition. The `TraverseIfCondition` also keeps track of both the if-scope and the else-scope since the different logical operators mutate these scopes. These operations function as described in Chapter 4. If the `TraverseIfCondition` reaches an invocation, then it tries to calculate the scope returned by the function by a `CalculateInvocationScope` which does all the necessary checks to see that the function is a check function and what values it checks. If the `CalculateInvocationScope` verifies that it is a check function, then it returns the restriction it guarantees to `TraverseIfCondition` which adds it to the proper context. After the calculations, the if-scope and else-scope are added to the respective context scopes making them available for the code inside the if statements.

5.5.1 Rolling If-Scope Calculations

When the code uses logical operators, then the if-condition traverses should give the current if scope should as part of the current known scope to the other part. This can be seen for instance in Figure 5.2 where the `p.FirstName` should not throw an error for the `FirstName` access.

```
1 Person p = new Person();
2 if (p != null && p.FirstName != null) // Should not throw error on p.FirstName,
3 { }                                   // since p can no longer be null.
```

Figure 5.2: Rolling if-scope calculation

Chapter 6

Evaluation

In this chapter, we describe the different kinds of problems and how many of the different problem types we found in the different evaluation projects. We describe the types of problems we found in the evaluation projects as well as some problems with try-catch error handling. We then present the results from the evaluation projects and the different statistics associated with them.

6.1 Code Problem Types

From the analysis of the CoreWiki and Innovation Norway Projects, three different types of faults started to arise. The first one was external validation errors, such as input model verification errors, where the input data from a user could result in unexpected behavior or exceptions. These faults could arise from a model not being validated before the code uses it. These faults are the once used by adversaries to try and take advantage of undefined behavior in applications.

The second form for faults is internal validation errors, where changes to the code could result in an exception, null exceptions, or incorrect behavior based on incorrectly using the existing code base. These types of faults can increase the development time since they arise from a lack of knowledge for the entire codebase. If more than one person is working on a project, or if the project starts to reach an unmanageable size, then these types of faults can become relevant.

The third type of faults has similarities with both the previous types of faults, which is the null errors. Languages like C#, Java, C, C++, Python and many more do not have facilities for defining if something can be null or not, which could result in undefined behavior or exception if something is assumed not to return null and it does. There were many null errors in the CoreWiki project,

but it is hard to say how many of them a user could trigger in the current code base without changes. The problem quickly arises if a developer expands the codebase and do not include the checks. If developers extend the codebase without the proper checks, then the code that was not problematic could quickly become so. ReDi resolves null errors by treating them as domain expansions of the variables.

6.2 Problems with Throwing Exceptions

There is a couple of problems with code throwing exception. One problem is that it could crash the software if the code does not handle the exception properly. Unhandled exceptions could result in loss of service or worst case, that a third party exploits the application in some way. Another problem with throwing exceptions is that it is very slow.

Figure 6.1 show the results from a simple test with a loop throwing exception and catching them, and one loop only adding the error to a list. The code is shown in Figure 6.2. The output in the figure shows that throwing exceptions is over 4000 times slower than just returning an error message, meaning that every throw takes 56.4μs in contrast to adding the error to the list taking only 12.7ns. Take note of the different prefixes used where throw uses microseconds and adding to list uses nanoseconds. These vast differences could quickly add up if a lot of the code throws exceptions instead of handling the error appropriately.

```
1 Testing iterations:                    1000000
2 Elapsed time for throw to list:     56402.1647 ms
3 Elapsed time for list:                 12.7133 ms
4 throw time / list time:              4436.4693
```

Figure 6.1: Results for throwing an error and adding to list, and only adding to list

Where this could be a problem is in a Denial of Service (DOS) attack. If a third party manages to trigger an exception in the web service, then a lot less computing power is needed to make the service run slower if the code uses exceptions.

6.3 CoreWiki

The CoreWiki project is an open source wiki project written in dotnet core by Jeffrey T. Fritz and the viewers of his streams at twitch.tv. The project is meant as a learning resource for developers to learn new technologies and APIs. We rewrote part of this project to include the helper library for better code analysis.

```
const int iterations = 1000000;
List<string> errors = new List<string>();
List<string> errors2 = new List<string>();
for (int i = 0; i < iterations; i++) {
    try {
        throw new Exception("An error occurred");
    } catch(Exception e) {
        errors.Add(e.ToString());
    }
}

for (int i = 0; i < iterations; i++) {
    errors2.Add("An error occurred");
}
```

Figure 6.2: Code throwing exceptions in a loop, and one loop only adding exception to list

In the CoreWiki project, ReDi helped with fixing both null checks and other inconsistencies.

One thing which ReDi provided, which is hard to detect otherwise was when a value has its domain reduced. In one case there is a function which turns a string into a URL friendly string. This function checked if the input parameter could be null, and then return null at once if it was. With the introduction of the `HasValue` attribute on the parameter, then this check was no longer necessary since the code should already have checked the value and we could remove the check from the code. The unit test for this function could also be modified since it was no longer any point in checking for null or an empty string. Testing for null and empty was unnecessary since these values are now impossible with the `HasValue` attribute. This code is also included in Figure 3.14 in Chapter 3.

We present the results in Figure 6.3 and shows that we found different kind of faults. The CoreWiki project relays on automatic-mapping of objects which makes it hard to apply the subtype system to the already defined types, but we rewrote some of them to show the potential. One of the input validation faults we found was not able to affect to program directly but had consequences found the software. The CoreWiki project would allow users to post an empty comment for an article, but the server did not validate them when received. This empty comment was sent towards the database and did not fail until it reaches the database, and it failed since the fields were empty. The input validation error which the code should have handled when received managed to throw a database error for that model validation. Error handling first at the database level is too late in the process since the server has now wasted many resources for mapping and transporting the message around, including throwing an error

which slows down the software further. The code also threw the error as an exception resulting in the slow down as described in Figure 6.1

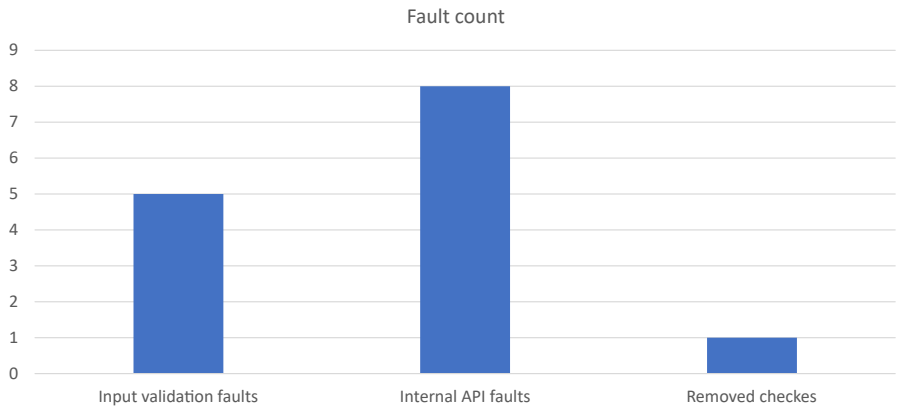


Figure 6.3: Results from CoreWiki

6.4 Innovation Norway Projects

The Innovation Norway projects are a set of projects developed by Bouvet. ReDi was applied to these projects to see if it was able to find code inconsistencies. The projects were obtained late in the development process of ReDi, resulting in not the same amount of attention has been used for analysis.

There were not many errors found in these projects by ReDi because of the limited time available, but we found some other inconsistencies. The most normal inconsistency shows one of the weaknesses of the C# language. The C# language is not easily able to define null checking. When we used ReDi at the code, it was possible to remove several redundant null checks since they no longer can be null. While removing redundant null checks would not have a significant impact on the performance of the application, removing null checks can help improving readability since there is less code to read. The extra attributes would also increase the descriptiveness of the code, potentially further increasing the readability of the code. As stated by Robert C. Martin in his book Clean Code [13], that the relationship between reading code and writing it is 10:1. Making the code easier to read could then improve the overall development time of the software. ReDi is also able to verify that a developer does not forget any null checks. Pointing out the missing null checks could result in reducing the development time for new and old code. The results are shown in Figure 6.4

As reflected in the results we found, the majority of the faults was internal

API errors as well as check removal. We found no validation results since these projects heavily relayed on reflections. Reflections make it hard to reason about the code since it relays on metaprogramming, and to be able to define rules for metaprogramming is outside the scope of this thesis.

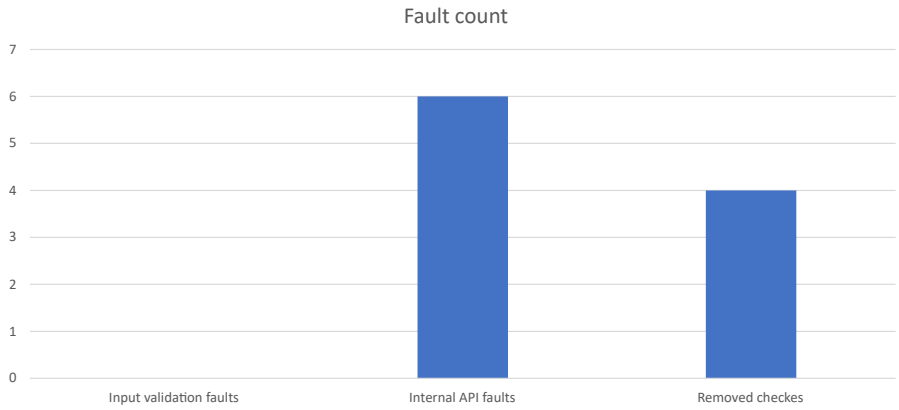


Figure 6.4: Results from Innovation Norway

6.4.1 Removal of Tests

One thing that ReDi help within the projects is ensuring consistencies. An example of this is in Figure 6.5 where there is a null check in the outer function as well as in the inner one, but not both are needed. In the example, a company is retrieved on line 2 and checked for null right afterward. ReDi has non-null by default, which results in the parameters for `CheckCompany` and `BuildDescription` cannot be null. The non-null parameter resulted in the removal of the `?` checks on line 12 and 20.

```
    public async CheckResponse CheckCompany(int companyId) {
        var company = await database.GetCompanyById(companyId);
        if (company == null)
            return new CheckResponse {Status = CheckStatus.CompanyNotFound};

        return CheckCompany(company);
    }

    private CheckResponse CheckCompany(Company company) {
        return new CheckResponse {
            TimeStamp = company?.Info?.LastUpdate;
    // New: TimeStamp = company.Info?.LastUpdate;
            Description = BuildDescription(company),
            Status = CheckStatus.CompanyFound
        };
    }

    private string BuildDescription(Company company) {
        return $@"{company?.Info?.NameAndAddress?.Name}
                org id: {company?.Info?.OrganizationId} \n";
// New: return £@"{company.Info?.NameAndAddress?.Name}
    }       //  org id: {company.Info?.OrganizationId} \n";
```

Figure 6.5: Example check removal from Innovation Norway. (Example is heavily modified from original code and only show the concept)

Chapter 7

Discussion

In this chapter, we discuss what ReDi can do and what differences there are to the other technologies for improving correctness. We also discuss future work and what could be possible to achieve with the same principles. Some of the possibilities are late model verification, view scope, immutability tracking, and the not null attribute.

7.1 Differences to Design by Contract Languages

The main differences to design by contrast languages are that C# was not designed for contracts. Both the mention languages, Eiffel and Dafny, was specially designed to be design-by-contract languages where C# was designed to be a general purpose language. In this thesis, we have shown that it is possible to use some of the same principles like model invariant and applying them to C# variables with the help of a tool. Dafny and Eiffel also is function centric meaning that they apply the main restrictions with the help of pre and post conditions on functions. ReDi, in contrast, to design by contract, only enforces pre and post conditions by specifying the required subtype of the input parameters and return type. In ReDi, the check function for the subtype can also always guarantee that the code fulfills a subtype.

7.2 Difference to Symbolic Execution

The difference between ReDi and symbolic execution tools like KLEE is that symbolic execution mostly detects paths that could lead to runtime exceptions.

Symbolic execution does not require any extra information for providing analysis. The problem is that developers do not have the option to provide intent information either, limiting the possible analysis. Having a combination of symbolic execution and the possibility to add extra intent information as well, we hope that ReDi can provide a richer analysis for the developer. Decisively limiting the scope of variables could also help with the problem of state explosion, were KLEE needs to explore too many different paths, and domains. In contrast, ReDi must limit the amount of analysis it can perform to the mentioned scope restrictions, because of its nature as a live analyzer. Where a symbolic execution tool can, in theory, run as long as needed, then ReDi needs to finish its analysis in a shorter time then the IDE updates the error reporting.

7.3 Where to use ReDi

ReDi has been good at detecting where the code is using null. It is also able to determine what return null with the help of the `Nullable` attribute. ReDi is also great at detecting places where there are unnecessary checks which are not needed. If developers add subtype attributes to the code, then ReDi can also help with finding missing checks and inconsistencies in the code, such as two classes having different expectations for the data.

7.4 Problems with Using the C# Language for ReDi

The C# programming language has much different syntax, which results in that it is hard for ReDi to analyze everything. This results in only the discovered coding patterns are given analysis and analyzed. There also must be written more analysis code for each new error type, resulting in the development time for ReDi increasing. We implemented an automatic child node analysis to give more analysis, even if the analysis of the main node was unsuccessful. C# also has a very developed reflection and metaprogramming system, which makes it harder to provide useful analysis for these scenarios. Reflections also make it possible to make generic implementations that work on more code, resulting in that it is hard for ReDi to verify the specific rules.

7.5 Restricted Relations

Design by Contract languages are applying restrictions to variables and parameters but are not able to create artificial subtypes. For instance, if we introduce the concept of meter and centimeter in code, then the available domain of them

are the same, but they have a relation to each other. The metric units usually do not restrict the domain of a variable, but the relation restricts the values. A relation restriction is when the code cannot directly convert between two units. For instance, meters and centimeters both have domains equal to all the real numbers, but a meter equals 100 centimeters, and we cannot directly assign them to a variable or parameter requiring centimeters before a conversion.

As we show in Figure 7.1 the area calculated would neither be centimeter squared or meter squared. The unit created by the multiplication would be hard to use since it is not a standard unit and could easily be misused. The circumference calculation in the figure, on the other hand, is incorrect since the circumference calculation requires the same unit. The examples from the figure would not throw an error in the C# language. With the help of the relation restrictions, it is possible to have the units of scientific calculations and formulas directly verified at compile time.

```
1  void Main() {
2      double length = 1.2/*m*/;        // Example of how units
3      double width = 52.4/*cm*/;       // could be applied
4
5      double area = length * width;    // area unit would be m*cm
6                                       // instead of m^2 or cm^2
7
8      double circumference
9          = length * 2 + width * 2;  // This does not make sense
10 }
```

Figure 7.1: Relation restriction

With compiler changes, it would also be possible to transform the units into a correct unit automatically. For instance, if a calculation tried to use meters and centimeters together, and the compiler knows of a transformation between them, then it could be automatically invoked on one of the values.

7.6 Problems and Considerations with ReDi

When using ReDi for a project, it is important that all team members who contribute code for the project use the tool. The enforcement is important because the C# compiler does not enforce verification of ReDi's annotations, and as such a developer that does not use ReDi could violate the code checking without being notified by a compiler or live analyzer. Automatic-build systems should also utilize ReDi when building the code for giving error messages in continuous integration workloads if a developer does not have the tool. We also wrote ReDi as a live analyzer, so the developer using ReDi is provides live

feedback while programming, which makes it easier and faster to develop the correct code.

7.7 Future Work

We present some future work which was not feasible to do in the time frame of the thesis in this section. The missing features include integration with late verification frameworks and especially the .net ComponentModel.DataAnnotation framework. The missing features also include the concept of view scopes to reduces the number of object mappings as well as immutability and the NonNull and subtype field requirement for object members. The last taking point is the advanced scope calculation.

7.7.1 Integration with ComponentModel.DataAnnotation

The dotnet base library contains a namespace called DataAnnotation, which the dotnet team created for model verification. The difference between DataAnnotation and ReDi is that DataAnnotation is late verification where ReDi is early verification. Early and late verification refers to in this context if verification happens before or after the code assigns a value to a field. Early requires that the check happens before the value assignment, while late verification happens after the assignment. We planned an extension to ReDi which allowed it to support late verification by assigning a subtype to an object that the code has verified late. Late verification only guarantees the restrictions on the object members if the main object has a verified subtype associated with it. It would also be better to use late verification on user input so that a map between user input and a verified object is not needed.

7.7.2 View Scopes

A use case for different scopes could be view scopes. A view scope would be a restriction of the visibility of the members of an object based on the scope. The scope would determine what members would be available, and what would not be available. A reason to have view scopes would be to not need to map objects between different implementations of them. A way to think of these view scopes would be an interface. An interface controls what on the object is available to call and what is not. A consideration with view scopes is that they would need to be mutable with only one thread access, or immutable.

A developer would need to use a transformation method for transferring one view scope to another view scope, to get access to different fields. The developer could also merge several objects in the transformation function to create a new object. View scopes result in the possibility to change the view scope of the

same object and expose different fields or members instead of having to create a new object. The view scopes would save processing since they do not need to perform the copying of fields.

Another use case of view scopes would be to define what keys exists in a dictionary so that an IDE can give better code completion. The C# dynamic type could also benefit from the same view scope by defining the available members on the dynamic type.

7.7.3 Immutability

Immutability, as a domain expansion, could be an interesting concept. The concept behind immutability restriction is it would be possible for a function to return an immutable object. The object is restricted from change after the function returns it, but the function could mutate the object before it returned it. Immutability would extend on the system with nullability since we would need to add an extension to the already defined subtype system to support it. Like nullability, immutability would also be a special case since it would modify if assignments are allowed or not.

The immutability restriction would allow an object to be locked down on some point in the code. Object locking could be useful if several functions are configuring an object, but when the functions finish, the object should be locked down and return to the caller function. C# has some functionality for this with the `readonly` modifier, but C# does not allow to arbitrary add the `readonly` modifier to a return type of a function. The `readonly` modifier in C# does not either support locking down an object and all its children as well. One problem with the immutability is tracking if one of the objects function tries to modify the values of the object itself, which would violate the immutability restriction.

7.7.4 Require NonNull Field

An extension we could make to ReDi could be a `NonNull` attribute, which required that some fields of an object were not null. We show an example of the non-null behavior in Figure 7.2. We could also apply the same principles as the `NonNull` attribute to the subtype system by requiring that a member of an object contained a specific field with a subtype. If ReDi could not guarantee these cases when analyzing the code, then it would report an error specifying that the restriction is not meet.

```
class Person
{
    [Nullable]
    public string FirstName {get;set;}
}

public static void Test([NonNull("FirstName")]Person person)
{
    person.FirstName.Trim(); // Should not return error message
}
```

Figure 7.2: NonNullAttribute declaration requiering fiels to not be null

7.7.5 Advanced Scope Calculations

Another extension for ReDi is to apply a more advanced type checking. What we mean about that is making ReDi able to analyze the different range check performed in an if-statement to calculate the scope of a variable better. Currently, ReDi only determines if the variable belongs to a check function or not. If ReDi was able to understand the proper domain of a variable with a range, then it could detect overlapping subtypes. If it could detect overlapping subtype, then ReDi could detect if a variable already has the required domain or a more restricted domain and circumvent the if-check entirely.

Chapter 8

Conclusion

When the compiler considers the domain of the variables, parameters, fields, and properties, then the tooling and IDE can provide a more productive developer experience. It is easier to understand the intent of the code since it has extra information describing the code and what it is supposed to do. It also makes it easier for the compiler to check the code more thoroughly and verify that the code is correct. The results found with ReDi shows that it is possible to improve the code correctness with the help of reducing and more clearly defining the domain boundaries. The scenarios where ReDi works best is when verifying the input values from a user and mapping them to internal types as well as validating that the code invokes internal API calls correctly. ReDi was also beneficial in detecting null errors and keeping track of what can be null and what cannot. The nullability system in ReDi do have some benefits over the C# version 8 nullability feature, where ReDi allows a method to be marked as a null check as well as giving the option to mark external library function with the `Nullable` attribute.

The restrictions also give the benefit of knowing which type-checks are necessary and which type-checks are redundant. This feature could improve the performance of the code, but also remove the redundant code, reducing the code a developer needs to read and maintain. The restrictions in ReDi also improve the consistency for the data values throughout the software from user input, through proceeding and to the data storage layer. The reason for the improvement is that all the parts operate on the same restrictions instead of the code only checking the restrictions at user input and data storage levels.

Using built-in check functionality and calculation domain restriction with static analysis works for keeping track of the variable domains. ReDi also improved the type checking and code intention with the inclusion of attributes in the code to describe the subtype to keep track of the calculated domain. The same principles could be used to implement design-by-contract patterns in existing

languages.

Restricting the available values for a variable reduces type related faults and type ambiguity.

Bibliography

[1] *A powerful and user-friendly binary analysis platform!: angr/angr*. original-date: 2015-08-06T21:46:55Z. June 5, 2019. URL: https://github.com/angr/angr (visited on 06/05/2019).

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: (), p. 16.

[3] *Dafny: A Language and Program Verifier for Functional Correctness*. Microsoft Research. URL: https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/ (visited on 06/08/2019).

[4] *Dafny is a verification-aware programming language: Microsoft/dafny*. original-date: 2016-04-16T20:05:38Z. Mar. 5, 2019. URL: https://github.com/Microsoft/dafny (visited on 03/05/2019).

[5] DEFCONConference. *DEF CON 23 - Shoshitaishvili and Wang - Angry Hacking: The next gen of binary analysis*. URL: https://www.youtube.com/watch?v=oznsT-ptAbk (visited on 06/05/2019).

[6] *Documentation*. May 21, 2019. URL: https://www.eiffel.org/documentation (visited on 06/05/2019).

[7] *Eiffel History*. URL: http://www.berenddeboer.net/eiffel/archive/halstenbach_eiffel_history.html (visited on 06/05/2019).

[8] Eiffel Programming Language. *Eiffel: An Overview*. URL: https://www.youtube.com/watch?v=zynTlO-72gc (visited on 03/05/2019).

[9] Nicolas Fløysvik. *Dotnet live analyzer for domain restricted types*. June 15, 2019. URL: https://bitbucket.org/Nicro950/nicroware.analzer.master/src/master/ (visited on 06/15/2019).

[10] Jeffrey T. Fritz. *A simple ASP.NET Core wiki that we are working on during live coding streams: csharpfritz/CoreWiki*. original-date: 2018-03-27T15:44:33Z. June 10, 2019. URL: https://github.com/csharpfritz/CoreWiki (visited on 06/13/2019).

[11] ecma international. *Standard ECMA-334 C# Language Specification*. Dec. 2017. URL: http://www.ecma-international.org/publications/standards/Ecma-334.htm (visited on 10/03/2019).

[12] mairaw. *Code Contracts*. URL: https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts (visited on 06/07/2019).

[13] Robert C. Martin, ed. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2009. 431 pp. ISBN: 978-0-13-235088-4.

[14] B. Meyer. "Applying 'design by contract'". In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 0018-9162. DOI: 10.1109/2.161279. URL: http://ieeexplore.ieee.org/document/161279/ (visited on 06/05/2019).

[15] Chris Newcombe et al. "How Amazon web services uses formal methods". In: *Communications of the ACM* 58.4 (Mar. 23, 2015), pp. 66–73. ISSN: 00010782. DOI: 10.1145/2699417. URL: http://dl.acm.org/citation.cfm?doid=2749359.2699417 (visited on 01/16/2019).

[16] *Null References: The Billion Dollar Mistake*. InfoQ. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare (visited on 06/09/2019).

[17] *Source code for the CodeContracts tools for .NET. Contribute to microsoft/CodeContracts development by creating an account on GitHub.* original-date: 2015-01-06T19:48:32Z. June 7, 2019. URL: https://github.com/microsoft/CodeContracts (visited on 06/07/2019).

[18] *TLA+ Video Course*. URL: https://lamport.azurewebsites.net/video/videos.html (visited on 11/14/2018).