FernUniversität in Hagen, Fachbereich Informatik
Lehrgebiet Programmiersysteme
Univ.-Prof. Dr. Friedrich Steimann

Bachelor's Thesis

Constrained-based diagnostics and code fixes for C# using Z3, LINQ and Roslyn

Ricardo Niepel, né Wickel

Matriculation: 7627157

Examiner & Supervisor: Univ.-Prof. Dr. Friedrich Steimann
Lehrgebiet Programmiersysteme
Fachbereich Informatik

For convenient searching and e-reading, an electronic version of this work is available at `https://github.com/RicardoNiepel/Z3.ObjectTheorem/docs`. The source code of the implementation described in this thesis can be found at `https://github.com/RicardoNiepel/Z3.ObjectTheorem/src`.

Contents

Abstract

The main topic is the demonstration and evaluation of constrained-based deep fixes with the help of Roslyn diagnostics and code fixes at an Entity Framework 6 code first model. Deep fixes are presented to the user inside Visual Studio 2015. Based on the ideas behind LINQ, C# Expressions are used as a constraint generation language for object theorems. The Microsoft Z3 Theorem Prover is used for constraint solving.

1 Introduction

"Entity Framework is Microsoft's recommended data access technology for new applications" [Mich]. It is possible to use three different development workflows with EF (Entity Framework): *Model First*, *Database First* and *Code First*. This will only be considered because the upcoming version of EF (Entity Framework 7) will only support *Code First*. In the words of [Mil14], "Code First is a bad name [...] calling it something like 'code-based modeling' would have been much clearer".

EF Code First is mostly based on conventions[1]. Thus the class and property design settles the object/relational mapping. In addition, many data annotations[2] exist which can be used to adjust the mapping configuration. In C# attributes are used as data annotations.

If the conventions and data annotations are not sufficient to cover special scenarios, it is also possible to manually adjust the mapping configuration with the *Fluent API*. This provides a programmatic interface to the mapping configuration (see [Mice] and [Micd]). The existing conventions can also be removed or custom conventions can be added [Micf]. "Precedence is given to configuration through the fluent API followed by data annotations and then conventions." [Micb]

The usage of conventions and data annotations are popular because of its simplicity. It also leads to the programming pattern *convention over configuration* which is the recommended approach for EF: "What this means is that code first will assume that your classes follow the conventions that EF uses. In that case, EF will be able to work out the details it needs to do its job. However, if your classes do not follow those conventions, you have the ability to add configurations to your classes to provide EF with the information it needs" [Micc].

The downside of using conventions and data annotations - also called metaprogramming - is, that it is harder to develop bug free programs. As in [Fer15, p. 1] described, for frameworks (in this situation EF) using this kind of metaprogramming it is only possible to find configuration errors during runtime.

[1] official name: Code First Conventions, see [Micp] and [Micb]
[2] official name: Code First Data Annotations, see [Micc]

1.1 Demarcation

The *Lehrgebiet Programmiersysteme* from *FernUniversität in Hagen* has introduced the idea of deep fixing in their paper *Dr. Deepfix*[Fer15]. According to them, "a deep fix is a fix that anticipates and fixes all consecutive errors to an arbitrary depth of recursion (at least the depth required to find the first all-in-one solution)." [Fer15, p. 12]

They implemented and evaluated this approach with the motivating example of a Java application which uses the Java Persistence API (JPA). This sample was used because JPA uses "Java annotations which instruct JPA implementations as to which entities and fields are to be mapped to the database in which way." [Fer15, p. 3] For the implementation prototype the Eclipse IDE and the choco constraint solver[3] was used.

The goal of this thesis is to demonstrate the usage of deep fixes within another environment. This environment includes another programming language, IDE (integrated development environment), syntax analyzer, constraint solver and application scenario. The runtime behavior and possible performance degradation during development are evaluated.

Programming Language

C#

IDE

Visual Studio 2015 Update 1

Syntax Analyzer

Roslyn

Constraint Solver

Z3, theorem prover from Microsoft Research, licensed under the MIT license

Application Scenario

Entity Framework 6 Code First & SQL Server 2014

[3] Choco - Java Library for Constraint Programming (http://www.choco-solver.org/)

1.2 Contribution

In addition to the usage of deep fixes within another environment a new way of *Automatic constraint generation from metamodel* (see 4.3) is shown and implemented. Research showed that this combination is a new approach, not been covered by other publications. The only similar publication and implementation related to this was presented by Bart de Smet in [Sme09a], [Sme09b] and [Sme09c] but only supports primitive types, lacks object and property support and does not support assumptions.

For constraint generation an object-oriented metamodel is used. In contrast to Object Constraint Language, LINQ (Language-Integrated Query) is used as a constraint generation language. This results in a *LINQ to Z3 Binding* to produce *Z3.ObjectTheorems* (requirements and implementation details can be found in 4.3 Automatic constraint generation from metamodel).

1.3 Procedure

Subsequently, the procedure is described.

Section 2: Basics

As a prerequisite for all subsequent steps a short introduction is provided for the important technologies involved.

Section 3: Application Scenario / Problem Statement

On the basis of an application scenario, the problem statement is described. The necessary, manual steps, are pointed out, without the results of this thesis.

Section 4: Implementation

4.1 Specification of an EF Program Elements Metamodel

Creation of an object-oriented metamodel for EF program elements for constraint generation.

4.2 Manual constraint generation from metamodel

Framework/rules for constraint generation from classes with properties and references.

4.3 Automatic constraint generation from metamodel

A general approach for constraint generation from classes (object-oriented programming): Creation of a generic object/LINQ to Z3 constraint generator using the rules from 4.2.

4.4 Specification of EF Constraints

Using the metamodel from step 1 and the LINQ to Z3 constraint generator from 4.2 to specify the EF constraints.

4.5 Automatic EF Metamodel Instantiation

Using Roslyn to analyze the C# syntax tree and instantiate the corresponding EF metamodel-

4.6 EF Constraint Checking and Fixing

Using all outcomes from above to automatically analyze, check and fix source code against EF constraints inside Visual Studio. Roslyn Diagnostics Analyzer and Roslyn Code Fixes are used to integrate the outcome from 4.5 into Visual Studio.

2 Basics

In this thesis multiple technologies are used and discussed. As a foundation, the most important ones are briefly explained in the following sub-chapters.

- The constraint modeling and generation is based on the ideas behind *LINQ (Language-Integrated Query).*
- Because the LINQ standard query operator methods do not fulfill all the requirements in this thesis, a combination of *custom query operator methods*, *standard query operator methods* and *Custom Expression Tree Parsing* with the help of an *Expression Tree Visitors* is used.
- For on the fly constraint generation, syntax analysis and automatic code fixes the *.NET Compiler Platform (codename "Roslyn")* is used.
- For theoretical explanations of various constraints, the *SMT-LIB 2.0 standard* is used. The *Z3 Theorem Prover* solves the constraints.

2.1 C# Expression Trees

Expression Trees in C# and Visual Basic "represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`" [Mici]. Code, which is represented by *Expression Trees*, can be compiled and executed.

Expression Trees can be created from *lambda expressions* or by using the `Expression` class. A *lambda expression* is an anonymous function that can also be used to create delegates. Expression trees can only be constructed from expression lambdas (also called single-line lambdas) and cannot be constructed from statement lambdas (or multi-line lambdas). In listing 1 on the following page the same expression tree is created in both ways.

At lines 13-22 the parsing of an expression tree is shown. A lot of type conversions is required because the properties (e.g. `Left` and `Right`) only return the super class `Expression`. To simplify parsing or modifications of expression trees, the *Expression Tree Visitor Pattern* can be used (see 2.2 Expression Tree Visitor Pattern for an introduction). At lines 25-29 the compilation and execution of an expression tree is listed.

Most expression trees are used to enable dynamic modification of executable code or to use lambda expressions for declarative and strong-typed specifications which can be parsed or executed against various data sources. These types of queries are called *LINQ queries*[4].

[4]see 2.3 LINQ (Language-Integrated Query) for an introduction

Listing 1: Expression Trees (excerpts from [Mici])

```
// Created from Lambda Expressions
Expression<Func<int, bool>> expr1 = num => num < 5;

// Created using the Expression class
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> expr2 = Expression.Lambda<Func<int, bool>>(
  numLessThanFive,
  new ParameterExpression[] { numParam });


// Parsing
ParameterExpression param = (ParameterExpression)expr2.Parameters[0];
BinaryExpression operation = (BinaryExpression)expr2.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
  param.Name, left.Name, operation.NodeType, right.Value);
// This code produces the following output:
// Decomposed expression: num => num LessThan 5


// Compiling and Executing
Func<int, bool> result = expr2.Compile();
Console.WriteLine(result(4));
// This code produces the following output:
// True
```

2.2 Expression Tree Visitor Pattern

"The Visitor pattern enables the definition of a new operation on an object structure without changing the classes of the objects." [PJ98, p. 1] It "describes a mechanism for interacting with the internal structure of composite objects that avoids frequent type casts or recompilation." [PJ98, p. 3] An *Expression Tree Visitor* is a specialized design pattern, which can be used for traversing, examining or copying an expression tree.

In listing 2 the concrete usage of the *Expression Tree Visitor Pattern* is the class `InnermostWhereFinder` which does a search through the whole expression tree to find the innermost `Where` method call. The class `InnermostWhereFinder` inherits from class `ExpressionVisitor` which can be found in listing 3 on the following page which is the recommended approach from Microsoft to work with expression trees. `ExpressionVisitor` is designed to be inherited. "If a sub-expression changes after it has been visited, for example by an overriding method in a derived class, the specialized visitor methods create a new expression that includes the changes in the sub-tree. Otherwise, they return the expression that they were passed. This recursive behavior enables a new expression tree to be built that either is the same as or a modified version of the original expression that was passed to Visit." [Micj]

Listing 2: Expression Tree Visitor: InnermostWhereFinder (excerpt from [Micq])

```
1  class InnermostWhereFinder : ExpressionVisitor
2  {
3    private MethodCallExpression innermostWhereExpression;
4
5    public MethodCallExpression GetInnermostWhere(Expression expression)
6    {
7      Visit(expression);
8      return innermostWhereExpression;
9    }
10
11   protected override Expression VisitMethodCall(MethodCallExpression expression)
12   {
13     if (expression.Method.Name == "Where")
14     innermostWhereExpression = expression;
15
16     Visit(expression.Arguments[0]);
17
18     return expression;
19   }
20 }
```

Listing 3: Expression Tree Visitor implementation (excerpt from [Micj])

```
public abstract class ExpressionVisitor
{
  ...
  protected virtual Expression Visit(Expression exp)
  {
    if (exp == null)
      return exp;

    switch (exp.NodeType)
    {
      ...
      case ExpressionType.Negate:
      case ExpressionType.NegateChecked:
      case ExpressionType.Not:
        return this.VisitUnary((UnaryExpression)exp);
      case ExpressionType.And:
      case ExpressionType.AndAlso:
      case ExpressionType.Or:
      case ExpressionType.OrElse:
        return this.VisitBinary((BinaryExpression)exp);
      ...
      case ExpressionType.Constant:
        return this.VisitConstant((ConstantExpression)exp);
      case ExpressionType.Parameter:
        return this.VisitParameter((ParameterExpression)exp);
      ...
    }
  }
  ...
  protected virtual Expression VisitUnary(UnaryExpression u)
  {
    Expression operand = this.Visit(u.Operand);
    if (operand != u.Operand)
    {
      return Expression.MakeUnary(u.NodeType, operand, u.Type, u.Method);
    }
    return u;
  }
  ...
  protected virtual Expression VisitConstant(ConstantExpression c)
  {
    return c;
  }
  ...
  protected virtual Expression VisitParameter(ParameterExpression p)
  {
    return p;
  }
  ...
}
```

Another concrete implementation of an *Expression Tree Visitor* is the class `AndAlsoModifier` from listing 4. It changes all occurrences of an `AndAlso` expression to an `OrElse` expression.

Listing 4: Expression Tree Visitor: AndAlsoModifier (excerpt from [Mick])

```
1  public class AndAlsoModifier : ExpressionVisitor
2  {
3    public Expression Modify(Expression expression)
4    {
5      return Visit(expression);
6    }
7
8    protected override Expression VisitBinary(BinaryExpression b)
9    {
10     if (b.NodeType == ExpressionType.AndAlso)
11     {
12       Expression left = this.Visit(b.Left);
13       Expression right = this.Visit(b.Right);
14
15       // Make this binary expression an OrElse operation instead of an AndAlso
             operation.
16       return Expression.MakeBinary(ExpressionType.OrElse, left, right,
             b.IsLiftedToNull, b.Method);
17     }
18
19     return base.VisitBinary(b);
20   }
21 }
```

2.3 LINQ (Language-Integrated Query)

"LINQ is a set of features introduced in Visual Studio 2008 that extends powerful query capabilities to the language syntax of C# and Visual Basic. LINQ introduces standard, easily-learned patterns for querying and updating data, and the technology can be extended to support potentially any kind of data store" [Micm]. Several LINQ providers are already included in the .NET Framework: query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections and also Entity Framework.

"LINQ bridges the gap between the world of objects and the world of data. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source" [Micl].

LINQ makes a query a first-class language construct in C# and Visual Basic to allow queries against strongly typed collections of objects by using language keywords and familiar operators. C# language features which are supporting LINQ, but can also be used separately (see [Mica]) are shown here:

- Query Expressions
- Implicitly Typed Variables
- Object and Collection Initializers
- Anonymous Types
- Extension Methods
- Lambda Expressions (Expression Trees)
- Auto-Implemented Properties

Two different syntaxes exist for writing LINQ queries: Query Syntax and Method Syntax. "The query syntax must be translated into method calls for the .NET common language runtime (CLR) when the code is compiled. These method calls invoke the standard query operators [...]. They can be called directly by using method syntax instead of query syntax" [Mico]. Like in listing 5 on the following page query and method syntax are semantically identical.

Listing 5: Query Syntax and Method Syntax in LINQ (excerpt from [Mico])

```
var numbers = new[] { 5, 10, 8, 3, 6, 12};

// Query syntax:
IEnumerable<int> numQuery1 =
  from num in numbers
  where num % 2 == 0
  orderby num
  select num;

// Method syntax:
IEnumerable<int> numQuery2 =
   numbers
  .Where(num => num % 2 == 0)
  .OrderBy(num => num);
```

There are different options to enable a data source for LINQ querying (excerpt from [Micg]):

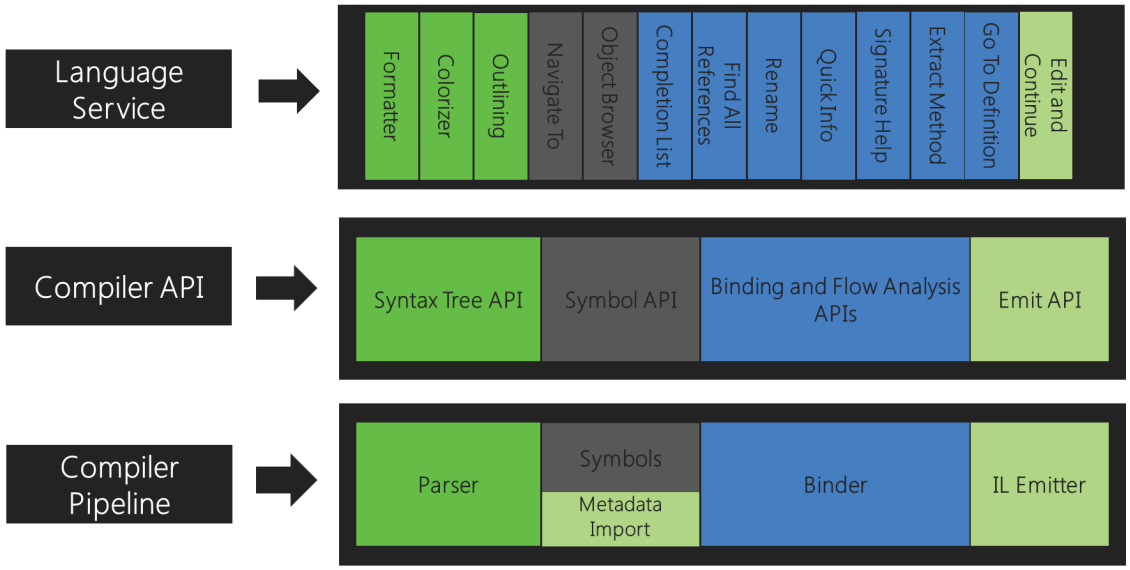
- Implementing the `IEnumerable<T>` interface in a type to enable LINQ to Objects querying of that type.
- Creating standard query operator methods such as `Where` and `Select` that extend a type, to enable custom LINQ querying of that type.
- Creating a provider for your data source that implements the `IQueryable<T>` interface. A provider that implements this interface receives LINQ queries in the form of expression trees, which it can execute in a custom way, for example remotely.
- Creating a provider for your data source that takes advantage of an existing LINQ technology. Such a provider would enable not only querying, but also insert, update, and delete operations and mapping for user-defined types.

2.4 .NET Compiler Platform ("Roslyn")

The new .NET Compiler Platform (hereinafter referred to with the codename "Roslyn") was released with Visual Studio 2015 at July 20, 2015. According to [Mcb], the core mission of Roslyn is to allow tools and end users to share in the wealth of information compilers have about the code. Instead of being opaque source-code-in and object-code-out translators, through Roslyn, compilers become platforms—APIs that can be used for code related tasks in tools and applications.

"To ensure that the public Compiler APIs are sufficient for building world-class IDE features, the language services that are used to power the C# and VB experiences in Visual Studio 2015 have been rebuilt using them. For instance, the code outlining and formatting features use the syntax trees, the Object Browser and navigation features use the symbol table, refactorings and Go to Definition use the semantic model, and Edit and Continue uses all of these, including the Emit API." [Mcb] as shown in figure 1.

Figure 1: Roslyn Compiler Pipeline, API and new Language Services



The Compiler API layer contains the object models that correspond with information exposed at each phase of the compiler pipeline, both syntactic and semantic. In the words of [Mcb], the most fundamental data structure exposed by the Compiler APIs is the syntax tree. "Syntax trees are the primary structure used for compilation, code analysis, binding, refactoring, IDE features, and code generation.

[...] Syntax trees represent the lexical and syntactic structure of source code. Although this information alone is enough to describe all the declarations and logic in the source, it is not enough information to identify what is being referenced. In addition to a syntactic model of the source code, a semantic model encapsulates the language rules [...] A semantic model represents all the semantic information for a single source file and can be used to discover the following:"

- The symbols referenced at a specific location in source.
- The resultant type of any expression.
- All diagnostics which are errors and warnings.
- How variables flow in and out of regions of source.
- The answers to more speculative questions.

The Compiler API layer also contains an immutable snapshot of the result of a single invocation of a compiler, including assembly references, compiler options, and source code files. During the analysis the compiler may produce a set of diagnostics covering everything from syntax, semantic, and definite assignment errors to various warnings and informational diagnostics.

Most of the existing diagnostics from tools like StyleCop or FxCop are migrated and can be found at [Mca]. An extensible API is exposed through the Compiler API layer which allows for user-defined analyzers to be plugged into a compilation to report diagnostics. These user-defined diagnostics are produced alongside compiler-defined diagnostics and are named *Roslyn Diagnostics Analyzer*.

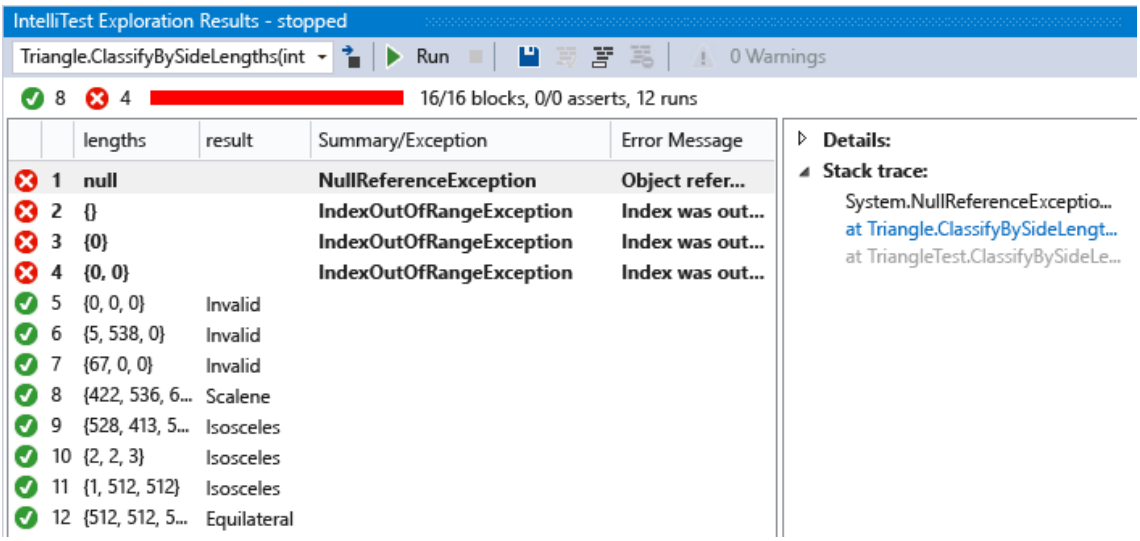
Producing diagnostics in this way has the benefit of integrating naturally with tools such as MSBuild and Visual Studio. Both depend on diagnostics for experiences such as halting a build based on policy and showing live squiggles in the editor and suggesting code fixes which are also called *Roslyn Code Fixes*.

2.5 Z3 Theorem Prover

Microsoft Research claims that Z3 is a state-of-the art theorem prover, [Res]. "It can be used to check the satisfiability of logical formulas over one or more theories. Z3 offers a compelling match for software analysis and verification tools, since several common software constructs map directly into supported theories." Microsoft Research also says that Z3 is a low level tool and it should be used as a component in the context of other tools that require solving logical formulas which is the aim of this thesis.

The best known extensions for previous Visual Studio versions which uses Z3 were Pex[5] and Code Digger[6]. They analyze .NET code to generate test data and a suite of unit tests, see figure 2.

Figure 2: Visual Studio 2015 IntelliTest



The Z3 input format is an extension of the one defined by the SMT-LIB 2.0 standard. Like line 1 in listing 6 on the next page constants of a given type can be declared, or function with multiple input and output parameters.

[5]Pex and Moles - Isolation and White box Unit Testing for .NET (http://research.microsoft.com/en-us/projects/pex/)

[6]Generate unit tests for your code with IntelliTest (https://msdn.microsoft.com/en-us/library/dn823749.aspx)

Listing 6: Z3 Input Sample

```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
(assert (> a 10))
(assert (< (f a true) 100))
(check-sat)
(get-model)
```

With the command `assert` a formula can be added into the Z3 internal stack. A "set of formulas in the Z3 stack is satisfiable if there is an interpretation (for the user declared constants and functions) that makes all asserted formulas true" [Res]. To test if all formulas are satisfiable, the command `check-sat` is used. The command `get-model` return an interpretation that makes all formulas on the Z3 internal stack true.

In this sample the constant `a` has the value `11` and the function `f` has the definition `(ite (and (= x!1 11)(= x!2 true))0 0)` in listing 7. The definition of `f` is based on `ite`'s (aka if-then-elses or conditional expressions), expressed in C# pseudocode in listing 8.

Listing 7: Z3 Model Sample

```
sat
(model
  (define-fun a () Int 11)
  (define-fun f ((x!1 Int) (x!2 Bool)) Int
    (ite (and (= x!1 11) (= x!2 true)) 0 0))
)
```

Listing 8: Z3 Model Sample: F as C# Pseudocode

```
public int f(int x1, bool x2)
{
  return x1 == 11 && x2 == true ? 0 : 0; // shorthand syntax
}

public int f(int x1, bool x2)
{
  if (x1 == 11 && x2)
  {
    return 0;
  } else {
    return 0;
  }
}
```

3 Application Scenario / Problem Statement

For the application scenario of this thesis the default configuration for EF 6.1.3 is used. This includes also all standard conventions. That means no existing conventions will be removed, no custom conventions will be added. The application scenario will purely focus at conventions and data-annotations. Programmatically configurations through *Fluent API* are not in the scope of this thesis."

3.1 Starting Point

For the following chapters, listing 9 on the following page shows the application scenario: The EF Code First model. The key scenario of this thesis is that the EF Code First model is syntactical correct and semantically incorrect. That means errors are not caught by the compiler or interpreter. This situation will maybe lead to crashes.

If the application starts, EF will initialize and check the current data model (= entity classes which will be used for object/relational modeling) against all conventions and data-annotations. If something is misconfigured, EF will throw an exception. Some misconfiguration can only be uncovered, if this model will be used for querying data and data manipulation. This is especially bad because in most situations no exceptions are thrown, e.g. the data is not saved into the database.

Listing 9: Application Scenario - EF Code First Model

```
public class Category
{
  public int CatNr { get; set; }

  [Index(IsUnique = true)]
  public string Name { get; set; }

  public virtual ICollection<Product> Products { get; set; }

  public ChangeInfo CreatedAt { get; set; }

  public virtual ICollection<ChangeInfo> UpdatedAt { get; set; }
}

public class Product
{
  [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public string Id { get; set; }

  public string Name { get; set; }

  public string Description { get; set; }

  public virtual ICollection<string> Tags { get; set; }

  public string ReplacementProductId { get; set; }

  public virtual Product ReplacementProduct { get; set; }

  public int CategoryID { get; set; }

  public virtual Category Category { get; set; }
}

public class ChangeInfo
{
  public DateTime Date { get; set; }

  public string User { get; set; }
}
```

If EF is initialized with the model in listing 9, a *ModelValidationException* is thrown, see listing 10 on the following page.

Listing 10: ModelValidationException

```
1 System.Data.Entity.ModelConfiguration.ModelValidationException:
2  One or more validation errors were detected during model generation:
3   EFScenario.Bad.Category:
4    EntityType 'Category' has no key defined.
5    Define the key for this EntityType.
6   EFScenario.Bad.ChangeInfo:
7    EntityType 'ChangeInfo' has no key defined.
8    Define the key for this EntityType.
```

3.2 Manual fixing with shallow fixes

Visual Studio as the primary IDE for .NET / C# development does not provide any error marking or automatic shallow fixes for EF Code First models. Shallow fixes "only solve one error without regarding consecutive errors" [Fer15, p. 2].

The only way to fix the reported errors is to use the exception message in listing 10 and manually improve the model. Several rounds of try and error are needed to fix all issues.

3.2.1 Round One

To fix the following error it is necessary to introduce meta-programming / attributes:

EFScenario.Bad.Category:
EntityType 'Category' has no key defined.
Define the key for this EntityType

This is required because the property naming `CatNr` violates the *Primary Key Convention*: "A property is a primary key if a property on a class is named ID (not case sensitive), or the class name followed by ID" [Micb].

The solution: the property `CatNr` on class `Category` is annotated with the `[Key]`-attribute, line 3 in listing 11 on the following page.

To fix the following error it is necessary to know the target database schema for the EF Code First model from listing 9 on page 18:

EFScenario.Bad.ChangeInfo:
EntityType 'ChangeInfo' has no key defined.
Define the key for this EntityType

In this application scenario `ChangeInfo` should be an EF *Complex Type*. *Complex Types* are no entities, thus they do not need a defined key property and the database mapping will not result in an additional table. The *Complex Types Convention* was not successful: "no primary key is registered [...]. Complex type detection also requires that the type does not have properties that reference entity types and is not referenced from a collection property on another type" [Micb].

The solution: the class `ChangeInfo` is annotated with the `[ComplexType]`-attribute, line 14 in listing 11.

Listing 11: EF Code First Model after round one of manual fixing

```
1  public class Category
2  {
3    [Key]
4    public int CatNr { get; set; }
5
6    ... // Unchanged in Round One
7  }
8
9  public class Product
10 {
11   ... // Unchanged in Round One
12 }
13
14 [ComplexType]
15 public class ChangeInfo
16 {
17   ... // Unchanged in Round One
18 }
```

3.2.2 Round Two

After compiling the new model from Round One EF throws a new exception at initialization, see listing 12 on the next page.

Listing 12: ModelValidationException after Round One

```
System.Data.SqlClient.SqlException:
 Column 'Name' in table 'dbo.Categories' is of a type that is invalid for use as
    a key column in an index.
```

To fix this error it is essential to know the EF type to database type mapping for strings. EF default type mapping maps all string properties to nvarchar(max) fields. In addition to this SQL Server has a maximum size of 900 bytes for index keys[7], therefore the default type mapping has to be changed.

The solution: the property `Name` on class `Category` is annotated with the `[MaxLength(255)]`-attribute, line 4 in listing 13.

Listing 13: EF Code First Model after round two of manual fixing

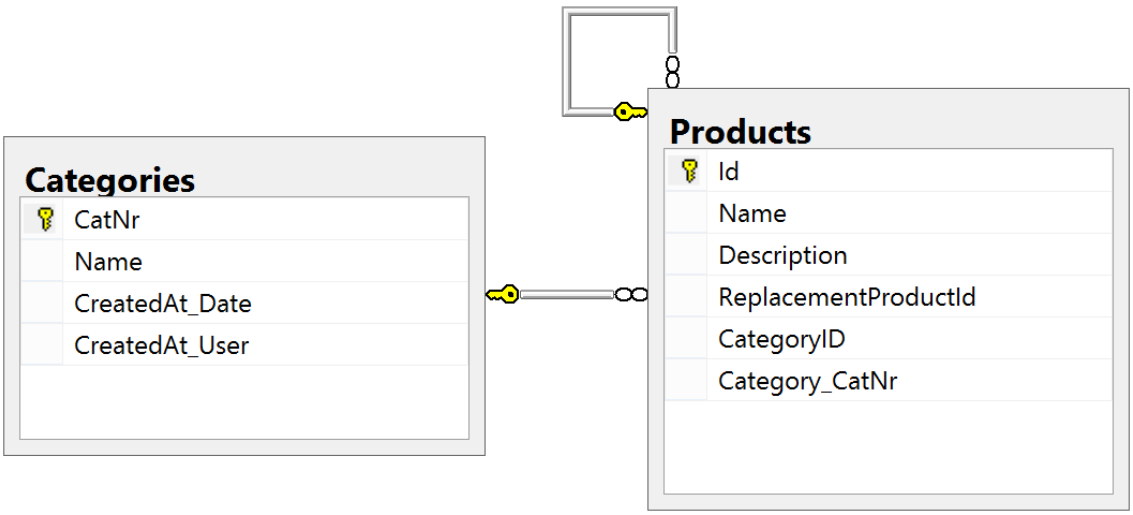
```
1  public class Category
2  {
3    [Index(IsUnique = true)]
4    [MaxLength(255)]
5    public string Name { get; set; }
6
7    ... // Unchanged in Round Two
8  }
9
10 public class Product
11 {
12   ... // Unchanged in Round Two
13 }
14
15 public class ChangeInfo
16 {
17   ... // Unchanged in Round Two
18 }
```

3.2.3 Round Three

After compiling the new model from Round Two EF throws no new exception at initialization and it seems as if everything is fine. But the complicated process of bug fixing without getting runtime exceptions just starts.

[7] https://technet.microsoft.com/en-us/library/ms191241.aspx

Figure 3: Database Diagram after round two of manual fixing



The database model at figure 3 was successfully created, but has differences compared with the current EF model:

1. Missing: `Category.UpdatedAt` and `Product.Tags`
2. Added: `Category_CatNr`

Both missing properties are ignored from EF. This is because EF cannot model collections of simple types (`ICollection<string> Tags`) and collections of complex types (`ICollection<ChangeInfo> UpdatedAt`). EF ignores these without throwing any exception. This is also the reason `ChangeInfo` needs to be annotate with the `[ComplexType]`-attribute (line 14 at listing 11 on page 20). To make this more obvious, it is a good practice to annotate both properties with the `[NotMapped]`-attribute, line 5 and 15 in listing 14 on the next page.

The added database column `Category_CatNr` has a special reason regarding the `NavigationPropertyNameForeignKeyDiscoveryConvention`: "Convention to discover foreign key properties whose names are a combination of the dependent navigation property name and the principal type primary key property name(s)" [Micn]. `Product.CategoryID` is not automatically mapped as the foreign key property because the primary key property is `CatNr` and not the default `ID`.

The solution: the property `Category` on class `Product` is annotated with the `[ForeignKey("CategoryID")]`-attribute in line 20 in listing 14 on the next page.

In addition to this, `[DatabaseGenerated(DatabaseGeneratedOption.Identity)]` is currently useless because it only changes the primary key column on certain types:

- Numbers, like `int` and `long`: EF creates an identity column
- `Guid`: EF creates an `uniqueidentifier` column with default value `newsequentialid()`

Using `[DatabaseGenerated(DatabaseGeneratedOption.Identity)]` at `string` types is ignored and EF does not throw any exception. In this application scenario it is a bug, it should be a `uniqueidentifier` column with default value `newsequentialid()`.

The solution: the type of property `Id` on class `Product` is changed to `Guid` in line 11 in listing 14.

Listing 14: EF Code First Model after round three of manual fixing

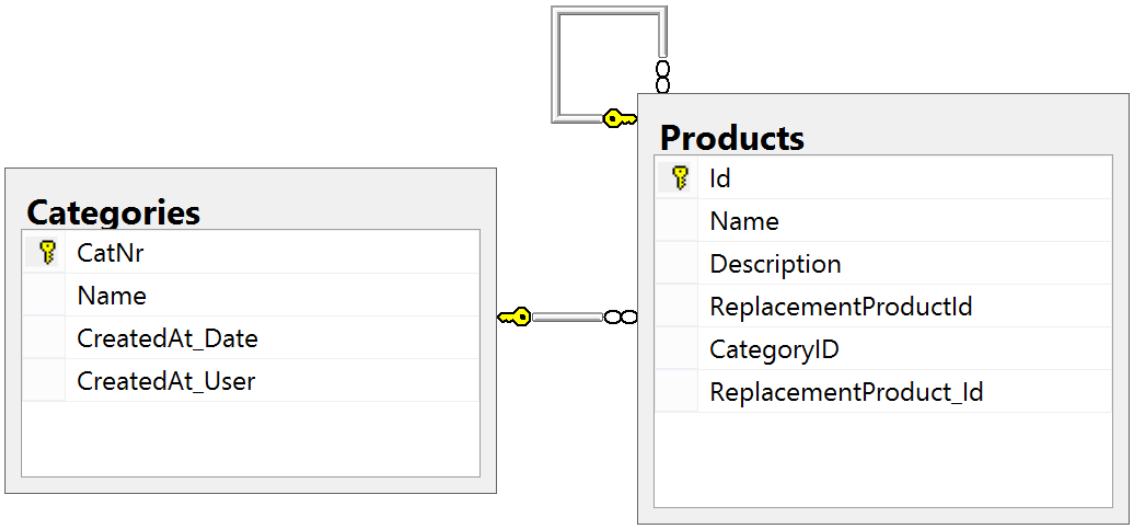
```
1  public class Category
2  {
3    ... // Unchanged in Round Three
4
5    [NotMapped]
6    public virtual ICollection<ChangeInfo> UpdatedAt { get; set; }
7  }
8
9  public class Product
10 {
11   public Guid Id { get; set; }
12
13   ... // Unchanged in Round Three
14
15   [NotMapped]
16   public virtual ICollection<string> Tags { get; set; }
17
18   ... // Unchanged in Round Three
19
20   [ForeignKey("CategoryID")]
21   public virtual Category Category { get; set; }
22 }
23
24 public class ChangeInfo
25 {
26   ... // Unchanged in Round One
27 }
```

3.2.4 Round Four

Compiling the model from Round Three and initializing EF is successful. However, the database model in figure 4 seems to have a new error:

- Added: `ReplacementProduct_Id`

Figure 4: Database Diagram after round three of manual fixing



The reason is the different type for property `string ReplacementProductId` compared to the primary key property type (`Guid Id`).

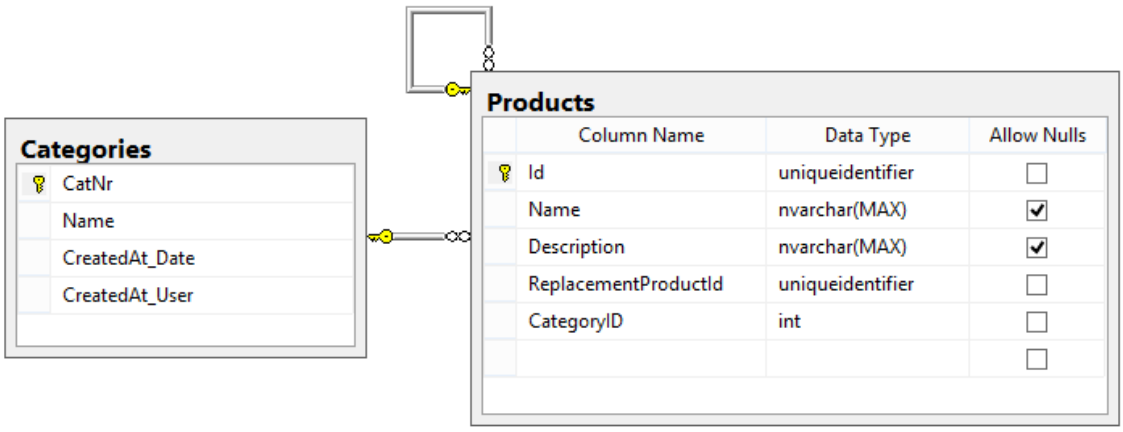
The solution: the type of property `ReplacementProductId` on class `Product` is changed to `Guid` in line 10 in listing 15 on the following page.

Listing 15: EF Code First Model after round four of manual fixing

```
1  public class Category
2  {
3    ... // Unchanged in Round Three
4  }
5
6  public class Product
7  {
8    ... // Unchanged in Round Three
9
10   public Guid ReplacementProductId { get; set; }
11
12   ... // Unchanged in Round Three
13 }
14
15 public class ChangeInfo
16 {
17   ... // Unchanged in Round One
18 }
```

3.2.5 Round Five

Figure 5: Database Diagram after round four of manual fixing



One last thing is still wrong. It is no longer automatically nullable because the type of `ReplacementProductId` has changed from a referenced type (`string`) to a value type (`Guid`). As seen in figure 4 on the previous page, this resulted in a required foreign key relationship (= not null).

To change it back to an optional foreign key relationship, the type of `ReplacementProductId` has to change to `Guid?`.

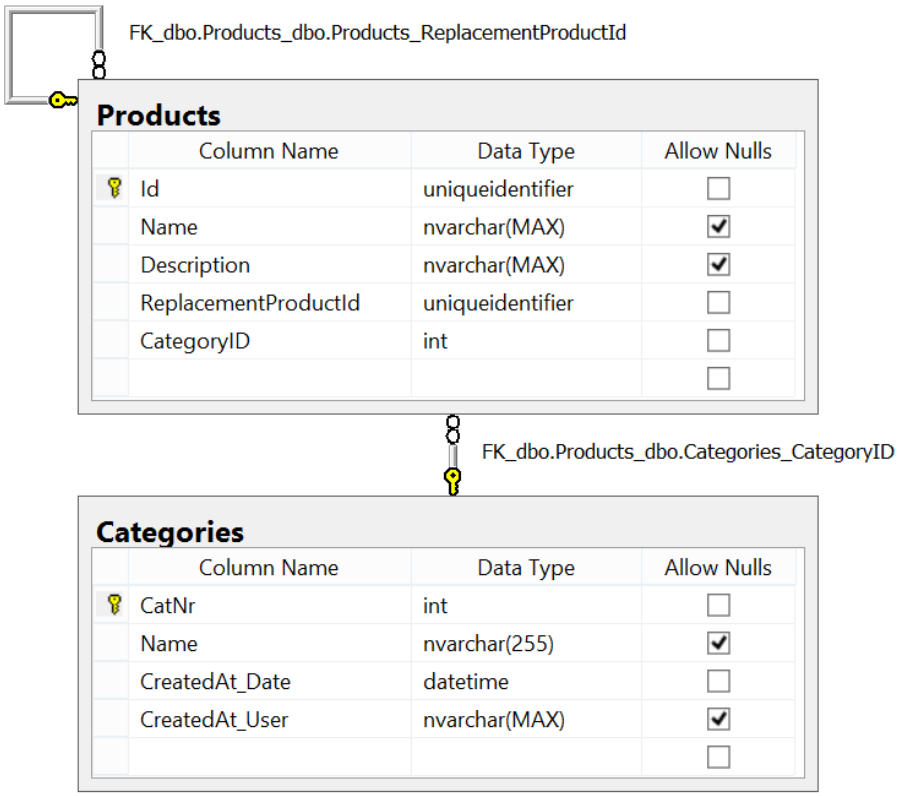
3.3 Conclusion

Five rounds of troubleshooting / try and error are necessary to fix this very short application scenario in listing 9 on page 18. The following challenges complicated the manual procedure:

- Only a small portion of errors lead to exceptions.
- Only a part of the errors is reported through exceptions, after fixing these, new exceptions are thrown.
- Errors are reported without regarding consecutive errors.
- A deep understanding of the EF conventions is necessary.
- A deep understanding of the .NET type system and its impact on EF is necessary.
- Cross-relationships regarding primary keys, foreign keys, foreign key relationships and complex types exist.

The database diagram of the fixed application scenario can be found in figure 6 and the belonging source code in listing 16 on the following page.

Figure 6: Database Diagram of fixed EF Code First Model



Listing 16: Fixed EF Code First Model

```
public class Category
{
  [Key]
  public int CatNr { get; set; }

  [Index(IsUnique = true)]
  [MaxLength(255)]
  public string Name { get; set; }

  public virtual ICollection<Product> Products { get; set; }

  public ChangeInfo CreatedAt { get; set; }

  [NotMapped]
  public virtual ICollection<ChangeInfo> UpdatedAt { get; set; }
}

public class Product
{
  [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public Guid Id { get; set; }

  public string Name { get; set; }

  public string Description { get; set; }

  [NotMapped]
  public virtual ICollection<string> Tags { get; set; }

  public Guid ReplacementProductId { get; set; }

  public virtual Product ReplacementProduct { get; set; }

  public int CategoryID { get; set; }

  [ForeignKey("CategoryID")]
  public virtual Category Category { get; set; }
}

[ComplexType]
public class ChangeInfo
{
  public DateTime Date { get; set; }

  public string User { get; set; }
}
```

4 Implementation

The following section describes the implementation of the solution for the application scenario / problem statement. The solution for automatic constraint generation from an object-oriented metamodel should also be usable for other scenarios.

An object-oriented metamodel is created for all EF program elements, called *EF metamodel*. This *EF metamodel* can be used for constraint modeling and generation. To be able to generate constraints from an object-oriented metamodel, rules for constraint generation from classes with properties and references are created.

These rules are used for automatic constraint generation from classes (object-oriented programming). For this, a general approach using LINQ and Expressions Trees is implemented, called *Z3.ObjectTheorem*. Using the *EF metamodel* and *Z3.ObjectTheorem* all required EF constraints to solve the application scenario are specified.

Roslyn is used to analyze the C# syntax tree and instantiate the corresponding *EF metamodel*. Roslyn Diagnostics Analyzer and Roslyn Code Fixes are used to integrate all outcomes from above to automatically analyze, check and fix source code against EF constraints inside Visual Studio.

4.1 Specification of an EF Program Elements Metamodel

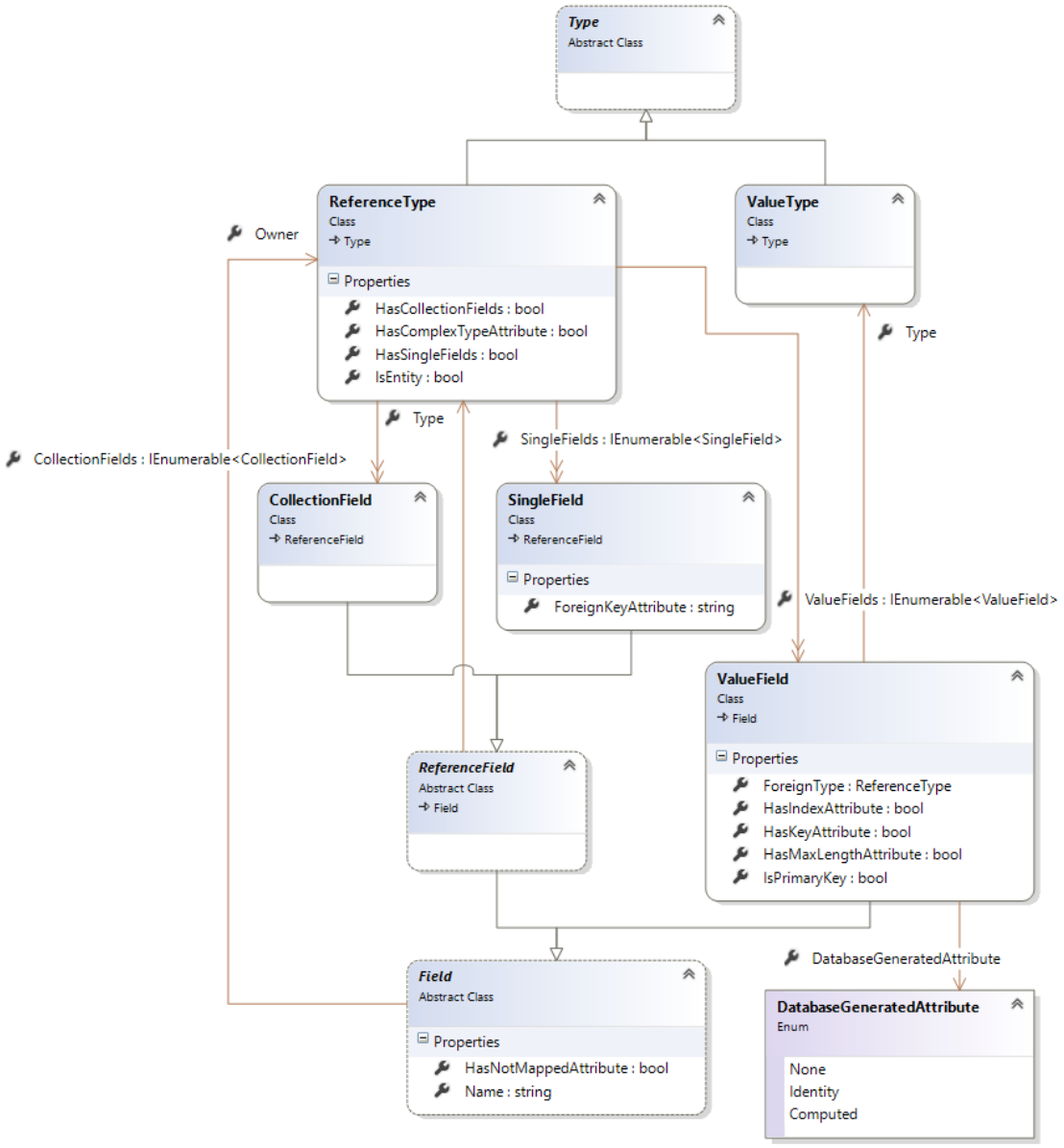
In accordance with [Fer15, p. 8] a similar metamodel can be defined to model all necessary EF program elements. The metamodel itself is object-oriented and is needed in further implementation steps.

Because in this thesis an EF Code First model is analyzed, the object-oriented model itself is very important for constraint generation. Thus typical elements from object-oriented programming languages like `Type`, `Field` and references between them are modeled as shown in figure 7 on the next page. Each essential characteristic from these elements are modeled as properties on these classes. For example, the name of a type is unimportant for EF because no convention is based on it. However, the name of a field is very important for several conventions. To leverage polymorphism when creating constraints, class hierarchies are used to differentiate between `ValueType` and `ReferenceType`, `ReferenceField` and `ValueField`, `CollectionField` and `SingleField`.

In addition to this, EF specific data-annotations (C# attributes) are added as properties. For example, the data-annotation for primary keys (= `KeyAttribute`) is modeled into a `HasKeyAttribute` boolean property on the `ValueField` class.

Some properties are only modeled to simplify the constraints. Thus `ValueField.IsPrimaryKey` mark a field as the primary key field and is only set by the primary key constraints. The properties `ReferenceType.HasCollectionFields` and `ReferenceType.HasSingleFields` are set within the automatic EF metamodel instantiation and are used to simplify other constraints.

Figure 7: EF Program Elements Metamodel



4.2 Manual constraint generation from metamodel

The metamodel from 4.1 Specification of an EF Program Elements Metamodel is object-oriented, therefore a structured approach is described for transforming a general object-oriented model into constraints. For explanation and as an example the EF program elements metamodel is used.

4.2.1 Transformation of classes and object instances

Each class is transformed into a new enumeration sort. All object instances get a unique identifier assigned. These identifiers are the enum values of the created class enumeration sort.

Sample

If three different instances of a `Field` exist, an enumeration sort named *Field_instances* with three enum values are defined as shown in listing 17.

Listing 17: Class & Object Transformation Sample

```
1 (declare-datatypes () ((Field_instances inst1 inst2 inst3)))
```

4.2.2 Transformation of class properties and their values

For each class property a function is created. The input parameter is an enum value of the corresponding class enumeration sort. The return value depends on the property type:

- Primitive Type: the corresponding primitive type
- Primitive Type Array: a set with the corresponding primitive type
- Complex Type: the corresponding class enumeration sort
- Complex Type Array: a set with the corresponding class enumeration sort

To assign the property values an assert is created for each property assignment.

Sample

The class `ReferenceType` from figure 7 on page 30 has the primitive type property `IsEntity` and the complex type array property `ValueFields`. For both properties a function is defined as shown in listing 18. Two property assignment samples are shown in line 7 and 8.

Listing 18: Property Transformation Sample

```
1 (declare-datatypes () ((ReferenceType_instances inst1 inst2)))
2 (define-sort SetOf_ValueField_instances (Array ValueField_instances Bool))
3
4 (declare-fun isEntity (ReferenceType_instances) Bool)
5 (declare-fun valueFields (ReferenceType_instances) SetOf_ValueField_instances)
6
7 (assert (= (isEntity inst1) true))
8 (assert (= (isEntity inst2) false))
```

4.2.3 Transformation of class hierarchies

A datatype sort is created for each super type. For all sub types a constructor with one parameter with the corresponding class enumeration sort is defined.

Sample

The classes `CollectionField` and `SingleField` from figure 7 on page 30 are both sub types from `ReferenceField`. For each sub type the corresponding constructor is created and assigned to the super type sort as shown in listing 19.

For each constructor a function is defined which upcasts the sub-type to the super-type. To be able to determine and utilize the sub-type, if the super type is used, also recognizer and downcast functions are needed.

Listing 19: Class Hierarchy Transformation Sample

```
1  (declare-datatypes () ((CollectionField_instances c1 c2 )))
2  (declare-datatypes () ((SingleField_instances s1 s2 )))
3
4  (declare-datatypes () ((ReferenceField
5   (CollectionField (c CollectionField_instances ))
6   (SingleField (s SingleField_instances ))
7  )))
8
9  ; Constructors / Upcast
10 (declare-fun ColField_instances2RefField (CollectionField_instances )
11   ReferenceField )
12 (declare-fun SinField_instances2RefField (SingleField_instances )
13   ReferenceField )
14
15 ; Accessors / Downcast
16 (declare-fun RefField2ColField_instances (ReferenceField )
17   CollectionField_instances )
18 (declare-fun RefField2SinField_instances (ReferenceField )
19   SingleField_instances )
20
21 ; Recognizers
22 (declare-fun IsCollectionField_instances (ReferenceField ) Bool )
23 (declare-fun IsSingleField_instances (ReferenceField ) Bool )
```

4.3 Automatic constraint generation from metamodel

The basic idea for using C# Expressions as a replacement for OCL (Object Constraint Language) is documented in detail from David H. Akehurst et al. in [Dav+08]. OCL itself is a declarative language for describing rules which apply to any metamodel that cannot otherwise be expressed by diagrammatic notation. In the past OCL was used by Friedrich Steimann et al. "as a Constraint Generation Language" in [US13], [Ste15] and [Fer15].

Because this thesis generates constraints from C# source code and deep fix C# source code, the considerations from David H. Akehurst [Dav+08] and Friedrich Steimann [US13] are combined to use "C# as a Constraint Generation Language".

More specifically the feature called LINQ (Language-Integrated Query) is used as a constraint generation language. LINQ is a fully integrated and strongly typed query language inside C#. If LINQ queries are used for constraint generation, they can also benefit from these advantages. The queries itself are expressed inside closures. "The concept of functions enclosed in other functions paired with an environment provided by the outer function is known as closure." [Dav+08, p. 3]. In C# this is called lambda expression.

The implementation itself consists of two phases: interpretation of the lambda expression tree and transformation into Z3 constraints and method calls. Within the second phase the results from *4.2 Manual constraint generation from metamodel* are used as a general approach for constraint generation from classes (object-oriented programming).

4.3.1 Interpretation of the Lambda Expression Tree

Bart de Smet already had the fundamental idea of using LINQ expression trees to build constraints ([Sme09a], [Sme09b], [Sme09c]). The following sample is based on [Sme09a] with further explanation and additional code samples.

The objective is to let the Z3 Solver solve an exclusive OR with two constants a and b, listing 20. The listing 21 shows the exact same objective with method calls in C# source code.

Listing 20: Sample Objective in Z3 input format

```
(declare-const a Bool)
(declare-const b Bool)
(assert (xor a b))
```

Listing 21: Sample Objective with method calls

```
var context = new Context();

var a = context.MkConst("a", context.MkBoolType());
var b = context.MkConst("b", context.MkBoolType());

var xor = context.MkXor(a, b);
```

If the sample is implemented with LINQ / lambda expression, it can be written as one line of C# source code with one expression. The expression is of type `Expression<Func<bool, bool>>`, as shown in listing 22 because the expression uses two constants of type `boolean`. `^` is the bitwise or logical XOR operator in C#, thus `(a ^ b)` is the expression itself. Expression trees can also be build up with static method call to the `Expression` class. If listing 21 and listing 23 is compared, both expression trees are very similar.

Listing 22: Sample Objective with Lambda Expression

```
Expression<Func<bool, bool>> expr = (a, b) => (a ^ b);
```

Listing 23: Sample Objective with Expression class

```
var a = Expression.Parameter(typeof(bool), "a");
var b = Expression.Parameter(typeof(bool), "b");

var xor = Expression.ExclusiveOr(a, b);
```

4.3.2 Transformation into Z3 Constraints / Method Calls

The design pattern *Expression Tree Visitor* can also be used to create a new expression tree, based on other expressions / frameworks: to create a constrained-based boolean expression tree for the Z3 solver. The only fundamental difference is, that each `Visit`-method returns an `Expr` (base class for Z3 expressions) instead of `Expression` as shown in listing 24. The creation of the Z3 expression can be found in line 8: `ctx.MkAnd((BoolExpr)a, (BoolExpr)b)`.

Listing 24: Expression Tree Visitor for Z3

```
1  ...
2  private Expr Visit(Context context, Dictionary<PropertyInfo, Expr> environment,
       Expression expression, ParameterExpression param)
3  {
4    switch (expression.NodeType)
5    {
6      case ExpressionType.And:
7      case ExpressionType.AndAlso:
8        return VisitBinary(context, environment, (BinaryExpression)expression,
       param, (ctx, a, b) => ctx.MkAnd((BoolExpr)a, (BoolExpr)b));
9  ...
10 }
11
12 private Expr VisitBinary(Context context, Dictionary<PropertyInfo, Expr>
       environment, BinaryExpression expression, ParameterExpression param,
       Func<Context, Expr, Expr, Expr> ctor)
13 {
14   return ctor(context, Visit(context, environment, expression.Left, param),
       Visit(context, environment, expression.Right, param));
15 }
16 ...
```

In the next step LINQ is used to chain multiple expressions together, thus type safety and compile-time checking is available. In listing 25 on the next page a new theorem with two integer symbols (=constants) are created and multiple constraints are applied. The associated input for the Z3 solver and the model after solving are presented in listing 26 on the following page.

Listing 25: Sample of LINQ to Z3

```
using (var ctx = new Z3Context())
{
  var theorem = from t in ctx.NewTheorem<Symbols<int, int>>()
                where t.X1 < t.X2 + 1
                where t.X1 > 2
                where t.X1 != t.X2
                select t;

  var result = theorem.Solve();
}
```

Listing 26: Created Z3 input format from LINQ to Z3

```
(declare-const X1 Int)
(declare-const X2 Int)

(assert (< X1 (+ X2 1)))
(assert (> X1 2))
(assert (not (= X1 X2)))

{X1 = 3, X2 = 4}
```

A first version of this *LINQ to Z3 Binding* was also build by Bart de Smet in [Sme09a], [Sme09b] and [Sme09c]. The author of this thesis implemented and published a complete and up to date version of it at `https://github.com/RicardoNiepel/Z3.LinqBinding` as a preliminary work for this thesis because the implementation was incomplete and also build for an old version of the Z3 solver. This also includes an implementation for a specialized theorem: Sudoku.

4.3.3 From LINQ to Z3 Binding to Z3.ObjectTheorem

The *LINQ to Z3 Binding* from `https://github.com/RicardoNiepel/Z3.LinqBinding` was only able to handle primitives (a bunch of boolean and/or integer constants). To use the *LINQ to Z3 Binding* to generate constraints from an object-oriented metamodel, a complete rewrite is necessary, subsequently called *Z3.ObjectTheorem*.

Requirements

The following requirements need to be met:

1. Support of value types: bool, int, long and enums
2. Support of reference type: custom classes
3. Support of reference type: strings
4. Support of navigation properties (type of another custom class)
5. Support of class instances
6. Support of class hierarchies
7. Support of Any(), All() and both also nested
8. Support of assertions
9. Support of assumptions

Test-Driven Development

The development of this *Z3.ObjectTheorem* was done test-driven. Thus, the use of the API was in focus during the design and implementation phases. In listing 27 an UnitTest shows the API to fulfill requirements 1, 2, 5, 7 and 8. The class instances are created in line 4 and 5 (requirement 2). Both classes have a boolean property called `IsValidA` or `IsValidB` (requirement 1). There are three assertions (requirement 8). The assertion in line 9 is a universal quantifier for all classes `ClassA` and `ClassB` (requirement 7). The other two assertions in line 10 and 11 are constraining specific instances of classes (requirement 5).

Listing 27: Simple UnitTest for Z3.ObjectTheorem

```
1  // Arrange
2  var objectTheorem = new ObjectTheoremContext();
3
4  var classAInstance = objectTheorem.CreateInstance<ClassA>("ClassAInstance");
5  var classBInstance = objectTheorem.CreateInstance<ClassB>("ClassBInstance");
6
7  // Act
8  objectTheorem.ConstraintBuilder
9      .AssertAll<ClassA, ClassB>((a, b) => a.IsValidA == b.IsValidB)
10     .Assert(() => classAInstance.IsValidA == true)
11     .Assert(() => classBInstance.IsValidB == false);
12
13 // Assert
14 var solved = objectTheorem.Solve();
15 Assert.IsNull(solved);
```

The interaction with the Z3 context is lazy-evaluated, that means only if the method `Solve` gets called, a new Z3 `Context` will be created with all the required datatypes, functions, assertions and assumptions. In listing 28 the generated Z3 input is shown. The complexity of transforming objects and expressions into constraints are hidden inside the *Z3.ObjectTheorem* framework. The transformation rules from *4.2 Manual constraint generation from metamodel* are used.

Listing 28: Z3 input generated from Z3.ObjectTheorem

```
1  (declare-datatypes () ((ClassA_instances ClassAInstance)))
2  (declare-fun IsValidA (ClassA_instances) Bool)
3
4  (declare-datatypes () ((ClassB_instances ClassBInstance)))
5  (declare-fun IsValidB (ClassB_instances) Bool)
6
7  (assert
8   (forall ((ClassA ClassA_instances)(ClassB ClassB_instances))
9   (= (IsValidA ClassA) (IsValidB ClassB))
10 ))
11 (assert (= (IsValidA ClassAInstance) true))
12 (assert (= (IsValidB ClassBInstance) false))
```

Support of reference type: strings

Because Z3 does not support `String` type, it is necessary to define all possible `String` values upfront, see line 4 in listing 29. *Z3.ObjectTheorem* use this information to generate a Z3 enumeration type in which each string value represents one enumeration constant.

Listing 29: Strings UnitTest for Z3.ObjectTheorem

```
1  // Arrange
2  var objectTheorem = new ObjectTheoremContext();
3
4  objectTheorem.SetPossibleStringValues("Hans", "Fred", "Max");
5
6  var classWithStringA =
       objectTheorem.CreateInstance<ClassWithStringA>("ClassWithStringA");
7
8  // Act
9  objectTheorem.ConstraintBuilder
10   .AssertAll<ClassWithStringA>(c => c.FirstName != "Hans")
11   .Assert(() => "Fred" != classWithStringA.FirstName);
12
13 // Assert
14 var solved = objectTheorem.Solve();
15 Assert.IsNotNull(solved);
16
17 string firstName = solved.GetValue(classWithStringA, i => i.FirstName);
18 Assert.AreEqual("Max", firstName);
```

Support of navigation properties (type of another custom class)

Z3.ObjectTheorem treated navigation properties the same way as value type properties. Thus it is possible to navigate inside the constraints, as shown at lines 11 and 12 in listing 30.

Listing 30: Navigation Properties UnitTest for Z3.ObjectTheorem

```
1  // Arrange
2  var objectTheorem = new ObjectTheoremContext();
3
4  var typeInstance1 = objectTheorem.CreateInstance<Category>("TypeInstance1");
5  var typeInstance2 = objectTheorem.CreateInstance<Category>("TypeInstance2");
6  var fieldInstance1 = objectTheorem.CreateInstance<Product>("FieldInstance1");
7
8  // Act
9  objectTheorem.ConstraintBuilder
10   .Assert(() => typeInstance1.IsHighlighted == false)
11   .Assert(() => fieldInstance1.Category == typeInstance2)
12   .Assert(() => fieldInstance1.Category.IsHighlighted == true);
13
14 // Assert
15 var solved = objectTheorem.Solve();
16 Assert.IsNotNull(solved);
17
18 bool IsEntity1 = solved.GetValue(typeInstance1, i => i.IsHighlighted);
19 bool IsEntity2 = solved.GetValue(typeInstance2, i => i.IsHighlighted);
20 Assert.IsFalse(IsEntity1);
21 Assert.IsTrue(IsEntity2);
```

Support of class hierarchies

To use class hierarchies, it is necessary to register the super types, if no instances are created from them (e.g. if using abstract super types), see line 4 in listing 31 for a registration sample. Super types can be used the same way as types without class hierarchy can be, see line 11. In this sample, the property `Speed` is defined at the super type `Vehicle` and can be used as expected inside different constraints, see line 12 and 13.

Listing 31: Class Hierarchy UnitTest for Z3.ObjectTheorem

```
1  // Arrange
2  var objectTheorem = new ObjectTheoremContext();
3
4  objectTheorem.RegisterSuperType<Vehicle>();
5  var bicycleInstance1 = objectTheorem.CreateInstance<Bicycle>("BicycleInstance1");
6  var bicycleInstance2 = objectTheorem.CreateInstance<Bicycle>("BicycleInstance2");
7  var carInstance1 = objectTheorem.CreateInstance<Car>("CarInstance1");
8
9  // Act
10 objectTheorem.ConstraintBuilder
11   .AssertAll<Vehicle>(v => v.Speed > 0)
12   .AssertAll<Bicycle, Car>((b, c) => b.Speed < c.Speed)
13   .Assert(() => bicycleInstance1.Speed == 10);
14
15 // Assert
16 var solved = objectTheorem.Solve();
17 Assert.IsNotNull(solved);
18
19 int bicycle1Speed = solved.GetValue(bicycleInstance1, i => i.Speed);
20 int bicycle2Speed = solved.GetValue(bicycleInstance2, i => i.Speed);
21 int carSpeed = solved.GetValue(carInstance1, i => i.Speed);
22
23 Assert.AreEqual(10, bicycle1Speed);
24 Assert.IsTrue(bicycle2Speed > 0);
25
26 Assert.IsTrue(carSpeed > 0);
27 Assert.IsTrue(bicycle1Speed < carSpeed);
28 Assert.IsTrue(bicycle2Speed < carSpeed);
```

Support of assumptions

In addition to create constraints, which are built by `Assert` and `AssertAll`, it is also possible to use assumptions. In line 10 and 11 at listing 32 an assumption is created and stored for later usage. If the theorem is unsatisfiable, each assumption can be checked if it is wrong (see line 17) and can be removed from the theorem (see line 19). If the solver is able to solve the theorem, the value of the property, which is used inside the assumption, can be retrieved as shown in line 24.

Listing 32: Assumption UnitTest for Z3.ObjectTheorem

```
1  // Arrange
2  var objectTheorem = new ObjectTheoremContext();
3
4  var carInstance1 = objectTheorem.CreateInstance<Car>("CarInstance1");
5
6  // Act
7  objectTheorem.ConstraintBuilder
8    .AssertAll<Car>(c => c.IsFast == true);
9
10 var assume1 = objectTheorem.ConstraintBuilder
11   .Assume(() => carInstance1.IsFast == false);
12
13 // Assert
14 var solved = objectTheorem.Solve();
15 Assert.AreEqual(Status.Unsatisfiable, solved.Status);
16
17 var assume1Result = solved.IsAssumptionWrong(assume1);
18 Assert.IsTrue(assume1Result);
19 objectTheorem.ConstraintBuilder.RemoveAssumption(assume1);
20
21 solved = objectTheorem.Solve();
22 Assert.AreEqual(Status.Satisfiable, solved.Status);
23
24 bool isFast1 = solved.GetValue(carInstance1, c => c.IsFast);
25 Assert.IsTrue(isFast1);
```

4.4 Specification of EF Constraints

The metamodel from 4.1 and the LINQ to Z3 constraint generator from 4.3 are used to specify the EF constraints. Subsequently, the individual constraints are explained.

The implementation and corresponding Unit Tests for this section can be found in the following projects at the appended source code:

- Z3.ObjectTheorem.EF
- Z3.ObjectTheorem.EF.UnitTests

C# does not have any `a implies b` conditional logical operator. If required, the equivalent `!a || b` is used. In C# `a ^ b` is equivalent to `a XOR b`.

4.4.1 Primary Key Constraints

The *Primary Key Constraints* from listing 33 are read as follows. Lines 2-17 state that all ReferenceTypes, which are Entities, need to follow one of the primary key Conventions. Either one of the ValueFields needs to have the name "Id" or "ID", or only one of the ValueFields has the KeyAttribute. Lines 18-25 state that all ReferenceTypes, which are Entities, need exactly one primary key field.

Listing 33: Primary Key Constraints

```
1  .AssertAll<ReferenceType>(t =>
2    !t.IsEntity
3    // Primary Key Convention: IdKeyDiscoveryConvention
4    ^ t.ValueFields.Any(f =>
5      (f.Name == "Id" || f.Name == "ID") && f.IsPrimaryKey
6    )
7    // Primary Key Convention: KeyAttributeConvention
8    ^ (t.ValueFields.All(fn =>
9      fn.Name != "Id" && fn.Name != "ID")
10     &&
11     t.ValueFields.Any(f1 =>
12       f1.HasKeyAttribute
13       && f1.IsPrimaryKey
14       && t.ValueFields.All(f2 => f2 == f1 || !f2.HasKeyAttribute)
15     )
16   )
17 )
18 .AssertAll<ReferenceType>(t =>
19   !t.IsEntity
20   || t.ValueFields.Any(f1 =>
21     f1.IsPrimaryKey && t.ValueFields.All(f2 =>
22       f2 == f1 || !f2.IsPrimaryKey
23     )
24   )
25 )
```

4.4.2 Complex Type Constraints

At listing 34 the *Complex Type Constraints* are read as follows. Line 1 states that all ReferenceTypes are either Entities or ComplexTypes. The later corresponds to the ComplexType attribute. Line 3 deals with the fact, that ComplexTypes do not have any PrimaryKeys defined. At line 5 it is checked, that all ComplexTypes are not allowed to have Collection or Single ReferenceFields.

Listing 34: Complex Type Constraints

```
1 .AssertAll<ReferenceType>(t => t.IsEntity ^ t.HasComplexTypeAttribute)
2 .AssertAll<ReferenceType>(t =>
3   !t.HasComplexTypeAttribute ^ t.ValueFields.All(vf => !vf.IsPrimaryKey))
4 .AssertAll<ReferenceType>(t =>
5   !t.HasComplexTypeAttribute ^ (!t.HasCollectionFields && !t.HasSingleFields))
```

4.4.3 String Index Constraints

The *String Index Constraints* from listing 35 are read as follows. Lines 2-4 state that if an IndexAttribute exists for a ValueField, it is necessary that the field type is not a string type or that the MaxLengthAttribute is used. Also it is not allowed to use the MaxLengthAttribute if no IndexAttribute is used. Line 5 states, that each ValueField cannot have the KeyAttribute and IndexAttribute simultaneously.

Listing 35: String Index Constraints

```
1 .AssertAll<ValueField>(vf =>
2   (!vf.HasIndexAttribute && !vf.HasMaxLengthAttribute)
3   ^ (vf.HasIndexAttribute && vf.Type != stringType)
4   ^ (vf.HasIndexAttribute && vf.HasMaxLengthAttribute))
5 .AssertAll<ValueField>(vf => !(vf.HasKeyAttribute && vf.HasIndexAttribute))
```

4.4.4 NotMapped Constraints

At listing 36 the *NotMapped Constraints* are read as follows. The NotMapped Attribute is required for all ValueType and ComplexType Collections. Thus in line 1-6 all collections of ValueType are iterated and if any ValueField is of these types it implies that it has the NotMapped attribute. In line 9 it is ensured that if the CollectionField type has a ComplexType attribute, the CollectionField itself has a NotMapped attribute.

Listing 36: NotMapped Constraints

```
1 foreach (var collectionOfValueType in collectionsOfValueType)
2 {
3   constraintBuilder
4     .AssertAll<ValueField>(vf =>
5      vf.Type != collectionOfValueType || vf.HasNotMappedAttribute);
6 }
7 constraintBuilder
8   .AssertAll<CollectionField>(cf =>
9     !cf.Type.HasComplexTypeAttribute || cf.HasNotMappedAttribute);
```

4.4.5 DatabaseGenerated Constraints

The *DatabaseGenerated Constraints* from listing 37 are read as follows. In the application scenario of this thesis line 2 ensures that the DatabaseGenerated attribute is only allowed to PrimaryKey fields. In addition to this, lines 7-12 deal with the allowed combinations: If the type is String, only the Computed options is allowed, if the type is Number, only Computed is not allowed, if the type is Guid, only Identity is not allowed.

Listing 37: DatabaseGenerated Constraints

```
1  .AssertAll<ValueField>(f =>
2    (!f.IsPrimaryKey && f.DatabaseGenerated == DatabaseGeneratedAttribute.None)
3    ^
4    (f.IsPrimaryKey
5      &&
6      (
7      (f.Type == stringType
8        && f.DatabaseGenerated == DatabaseGeneratedAttribute.Computed)
9      ^ (f.Type == numberType
10       && f.DatabaseGenerated != DatabaseGeneratedAttribute.Computed)
11     ^ (f.Type == guidType
12       && f.DatabaseGenerated != DatabaseGeneratedAttribute.Identity)
13     )
14   )
15 );
```

4.4.6 ForeignKey Constraints

The *ForeignKey Constraints* should be generated for the following conventions:

- ForeignKey Discovery Convention
- ForeignKey Type Matching
- ForeignKey optional relationship based on type

As part of this thesis it was not possible to find an appropriate solution for this. The main cause of this is the absence of String type support inside the Z3 Solver. All ForeignKey conventions inside EF use string options inside attributes and string concatenations to determine the ForeignKeys. For example, the ForeignKey attribute itself is not defined for the ForeignKey field but defined for the ReferenceField and uses a string option to correlate the ForeignKey field.

4.5 Automatic EF Metamodel Instantiation

Roslyn is used to analyze the C# project and instantiate the corresponding EF metamodel. The EF metamodel instantiation is based on the Roslyn semantic model, thus no low level syntax tree parsing is needed - Roslyn handles it internally.

The implementation and corresponding Unit Tests for this section can be found in the following projects at the appended source code:

- Z3.ObjectTheorem.EF.RoslynIntegration
- Z3.ObjectTheorem.EF.RoslynIntegration.Tests

Starting point for the analysis is the class which inherits from `DbContext`. This class is also the starting point for EF to find out which classes inside a project are modeled to be used with EF (these classes are also called entities). This `DbContext` class models each Entity with a property of base type `DbSet`, see listing 38.

Listing 38: EF Entitiy Discovery

```
1  var allTypeSymbols = compilation
2    .GetSymbolsWithName(s => true, SymbolFilter.Type).Cast<INamedTypeSymbol>();
3
4  dbContextTypeSymbol = allTypeSymbols.Single(t => t.BaseType.Name == "DbContext");
5  var entityTypeSymbols = dbContextTypeSymbol
6    .GetMembers()
7    .Where(m => m.Kind == SymbolKind.Property)
8    .Cast<IPropertySymbol>()
9    .Where(p => p.Type.Name == "DbSet")
10   .Select(p => p.Type)
11   .Cast<INamedTypeSymbol>()
12   .Select(t => t.TypeArguments[0])
13   .Cast<INamedTypeSymbol>();
```

For each entity and property, the corresponding constraints and assumptions are generated. In listing 39 the generation for ValueFields is shown. For example, for each ValueField the type is created or discovered (line 7) to use it for instantiation (line 16). The class hierarchies inside the EF metamodel are completely used for the different constraints (line 25 and 26). The `AssumptionHandler` has the responsibility to add the assumptions for the different types.

Listing 39: Constraint Generation for Properties

```
1  private void GenerateConstraintsForProperty(...)
2  {
3    Field field;
4    ...
5    if (propertySymbol.Type.IsValueType || "String" == typeName)
6    {
7      ValueType valType = GetOrAddValueType(typeName);
8
9      var valueField = _objectTheorem
10       .CreateInstance<ValueField>($"{typeSymbol.Name}{name}{ValueFieldPostfix}");
11
12     field = valueField;
13     valueFields.Add(valueField);
14
15     _constraintBuilder
16       .Assert(() => valueField.Type == valType);
17
18     _assumptionHandler
19       .HandleValueField(valueField, propertySymbol, propertyAttributes);
20   }
21   else
22   ...
23
24   _constraintBuilder
25       .Assert(() => field.Name == name)
26       .Assert(() => field.Owner == referenceType);
27
28   _assumptionHandler.HandleField(field, propertySymbol, propertyAttributes);
29 }
```

In this application scenario only model elements, which are not used inside the object oriented language, are modeled as assumptions: The EF data annotations / C# attributes. For each assumption the current class or property attributes are queried and the assumptions are generated, see listing 40. All assumptions are implemented as distinct classes to be able to use them later inside the automatic code fixing. As shown in listing 41 these classes also include a ranking which will be used to define the deletion order of wrong assumptions.

Listing 40: Assumption Generation

```
var hasKeyAttribute = propertyAttributes
  .Any(a => a.AttributeClass.Name == "KeyAttribute");
var assumption = _constraintBuilder
  .Assume(() => valueField.HasKeyAttribute == hasKeyAttribute);
_propertyAssumptions.Add(
  new HasKeyAttributePropertyAssumption(classProperty, assumption, valueField));
```

Listing 41: Assumption Class

```
public class HasKeyAttributePropertyAssumption
    : PropertyAssumption<ValueField, bool>
{
  ...

  public override Expression<Func<ValueField, bool>> PropertyExpression
      => vf => vf.HasKeyAttribute;

  public override int Rank { get; } = 1;

  protected override IEnumerable<string> GetAttributesToAdd(bool result)
  {
    if (result)
    {
      yield return "Key";
    }
  }

  protected override IEnumerable<string> GetAttributesToDelete(bool result)
  {
    if (!result)
    {
      yield return "Key";
    }
  }
}
```

4.6 EF Constraint Checking and Fixing

Roslyn Diagnostics Analyzer and Roslyn Code Fixes are used to automatically analyze, check and fix source code against EF constraints inside Visual Studio 2015. The implementation and corresponding Unit Tests for this section can be found in the following projects at the appended source code:

- Z3.ObjectTheorem.EF.Analyzer
- Z3.ObjectTheorem.EF.Analyzer.Tests
- Z3.ObjectTheorem.EF.Analyzer.Vsix

Inside the class `Z3ObjectTheoremEFAnalyzer` for each Roslyn compilation the EF Entities are discovered and the metamodel is instantiated. If the EF theorem is unsatisfiable, wrong assumptions are deleted and it is retried to be satisfiable.

The order of wrong assumption deletion is depending on the assumption rank from listing 41 on the preceding page. Assumptions with very strict constraints, e.g. `HasKeyAttributePropertyAssumption`, are deleted first. Assumptions with very loosely constraints, e.g. `HasNotMappedAttributePropertyAssumption`, are deleted at the end. This prevents a satisfiable model, where too many properties have the `NotMapped` attribute.

If all assumptions are deleted and the theorem is still unsatisfiable, an Error Diagnostic is reported, that no solution exist (figure 8). If the analyzer finds a satisfiable model, a Warning Diagnostic is reported and the user can choose to fix it automatically (figure 9).

Figure 8: Z3ObjectTheoremEFAnalyzer reports an error

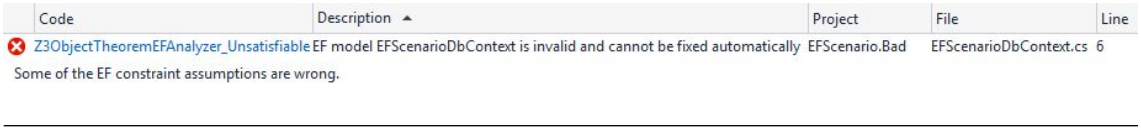
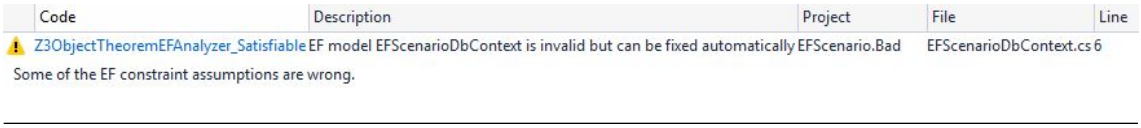


Figure 9: Z3ObjectTheoremEFAnalyzer reports a warning



Because Roslyn Diagnostics Analyzer and Roslyn Code Fixes are separated from each other, only key/value-pairs of strings can be used for additional information. To make it possible to reuse the same `ObjectTheorem` from the Diagnostics Analyzer for the associated Code Fix, an in-memory cache is used.

The class `Z3ObjectTheoremEFAnalyzerCodeFixProvider` uses the class `CSharpSyntaxRewriter` which is one of the internal Roslyn implementations of the *2.2 Expression Tree Visitor Pattern*. The base class `CSharpSyntaxRewriter` was inherited from class `AttributeAssumptionsRewriter`. Inside this class, for each `ClassDeclarationSyntax` and `PropertyDeclarationSyntax` the associated assumptions are evaluated against the ObjectTheorem model and the corresponding attributes are added and deleted.

Listing 42: Rewrite of ClassDeclarationSyntax based on Assumptions

```
1  public override SyntaxNode VisitClassDeclaration(ClassDeclarationSyntax node)
2  {
3    ...
4
5    if (classAssumptions.Count == 0)
6    {
7      return base.VisitClassDeclaration(node);
8    }
9
10   var newClass = (ClassDeclarationSyntax)base.VisitClassDeclaration(node);
11
12   foreach (var classAssumption in classAssumptions)
13   {
14     foreach (string attributeToDelete in
15         classAssumption.GetAttributesToDelete(_objectTheoremResult))
16     {
17       newClass = RemoveAttribute(newClass, attributeToDelete);
18     }
19
20     foreach (string attributeToAdd in
21         classAssumption.GetAttributesToAdd(_objectTheoremResult))
22     {
23       newClass = EnsureAttribute(newClass, attributeToAdd);
24     }
25   }
26
27   return newClass;
28 }
```

Inside Visual Studio 2015 a preview of the Code Fix which `Z3ObjectTheoremEFAnalyzerCodeFixProvider` generates is displayed (figure 10). It is also possible to see all changes in all files (figure 11). It is currently not possible to add any options to this dialogs and windows. Thus the first satisfiable model is used and a *Solution Space Explorer* like in [Fer15, p. 18] is not possible with the built-in options of Roslyn.

Figure 10: Roslyn CodeFix Preview Window

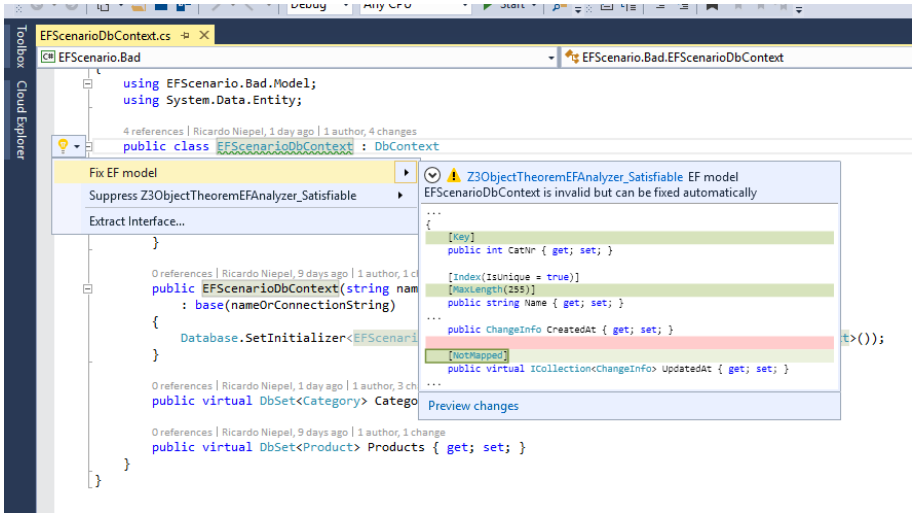
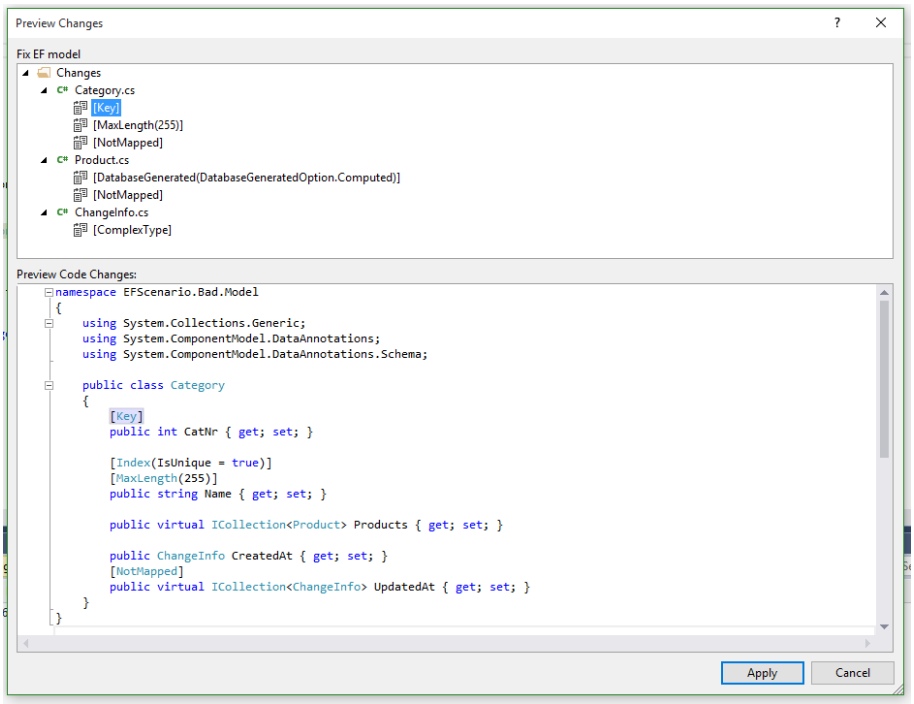


Figure 11: Roslyn CodeFix Preview Window



5 Evaluation

Roslyn Diagnostics Analyzers are running in the background of Visual Studio 2015. Thus slow analysis times are not blocking the user interface and the developer can work. During typing there will be drawn lines under the reported diagnostic positions (e.g. a class or property). But if the user know that specific *Roslyn Diagnostics Analyzers* are running in the background, he will expect that they show up quickly after he introduced a bug. If he needs to wait for several minutes after typing to see issues are shown up, this will not improve the development speed.

That is the reason why it is important to evaluate the runtime behavior of the solution of this thesis and to exclude a performance degradation during development. Therefore, the Z3.ObjectTheorem for Entity Framework applications as well as Z3.ObjectTheorems in general are measured. It is also necessary to evaluate the development productivity of the development of an object theorem itself with *C# Expressions*.

5.1 Z3.ObjectTheorem for Entity Framework

For evaluating Z3.ObjectTheorem for Entity Framework the application scenario and a created EF code first model from the official Microsoft SQL Server Sample Database AdventureWorksLT [8] are measured. At the AdventureWorksLT EF model some classes were deleted to experiment with different EF model sizes. At table 1 on the next page the results with a quad core CPU of 2.7 GHz are shown.

[8] http://msftdbprodsamples.codeplex.com/

Table 1: Statistics

	Application Scenario	Adventure Works LT 1	Adventure Works LT 2	Adventure Works LT 3
Class Count	3	5	6	8
Property Count	15	51	60	101
Assertion Count	67	180	209	348
Assumption Count	48	229	276	441
Constraint Count	115	415	**492**	**801**
Generation Time	00:00.004	00:00.023	00:00.028	00:00.041
Solving Rounds	7	8	7	9
Sat Solving Time	00:00.118	00:07.715	**00:14.887**	**00:57.167**
Unsat Avg Time	<00:01	00:01	00:02	00:07
Unsat Max Time	<00:01	00:03	00:03	00:20

Class Count defines the total class count inside the ObjectTheorem, independently if it is an *Entity* or *ComplexType*. The *Property Count* row has the total property count of all classes. *Assertion Count* and *Assumption Count* define the counts of individual `Assert` and `Assume` method calls at the ObjectTheorem. *Constraint Count* indicates the total constraint count inside the solver, thus this is the count after the simplifying process inside the solver. *Generation Time* defines the time needed to evaluate the Expression Trees and generate Z3 constraints from it. Multiple assumptions need to be deleted, after the solver finds a satisfiable model because every scenario is at the first run unsatisfiable. *Solving Rounds* represents the count of rounds to find the satisfiable model. *Sat Solving Time* defines the pure solver time of the satisfiable model, thus no constraint generation or assumption deletion is in it. For reference *Unsat Avg Time* and *Unsat Max Time* shows the average and maximum times during the unsatisfiable rounds.

With an increasing constraint count the solver needs disproportionately more time to solve the theorem. While the time of generation of constraints from expressions to the total constraint count and the property count to constraint count is linear, the constraint count to the needed solving time seems to be exponentially. A EF model with only 8 classes and 101 properties in total (12 per class) seems very small in today's software applications. Thus the usage of Z3 as the solver also seems to be not possible for real applications with this application scenario.

5.2 Z3.ObjectTheorem in General

To make sure, that the specific EF constraints and models are the reason for the exponential solving times from *Z3.ObjectTheorem for Entity Framework* (see 5.1) a different test scenario is created. In this scenario a EF unaware model is used and all assumptions are removed. So this Z3.ObjectTheorem models what the compiler also knows: Which classes with which properties with which types exist.

The class from listing 43 with exact same property count and layout is created multiple times. To automate this step, a T4 template[9] is used. The class defines properties for all field types: ValueFields for primitives and for collections of primitives, ReferenceField with a CollectionField.

Listing 43: Generic Scenario Class

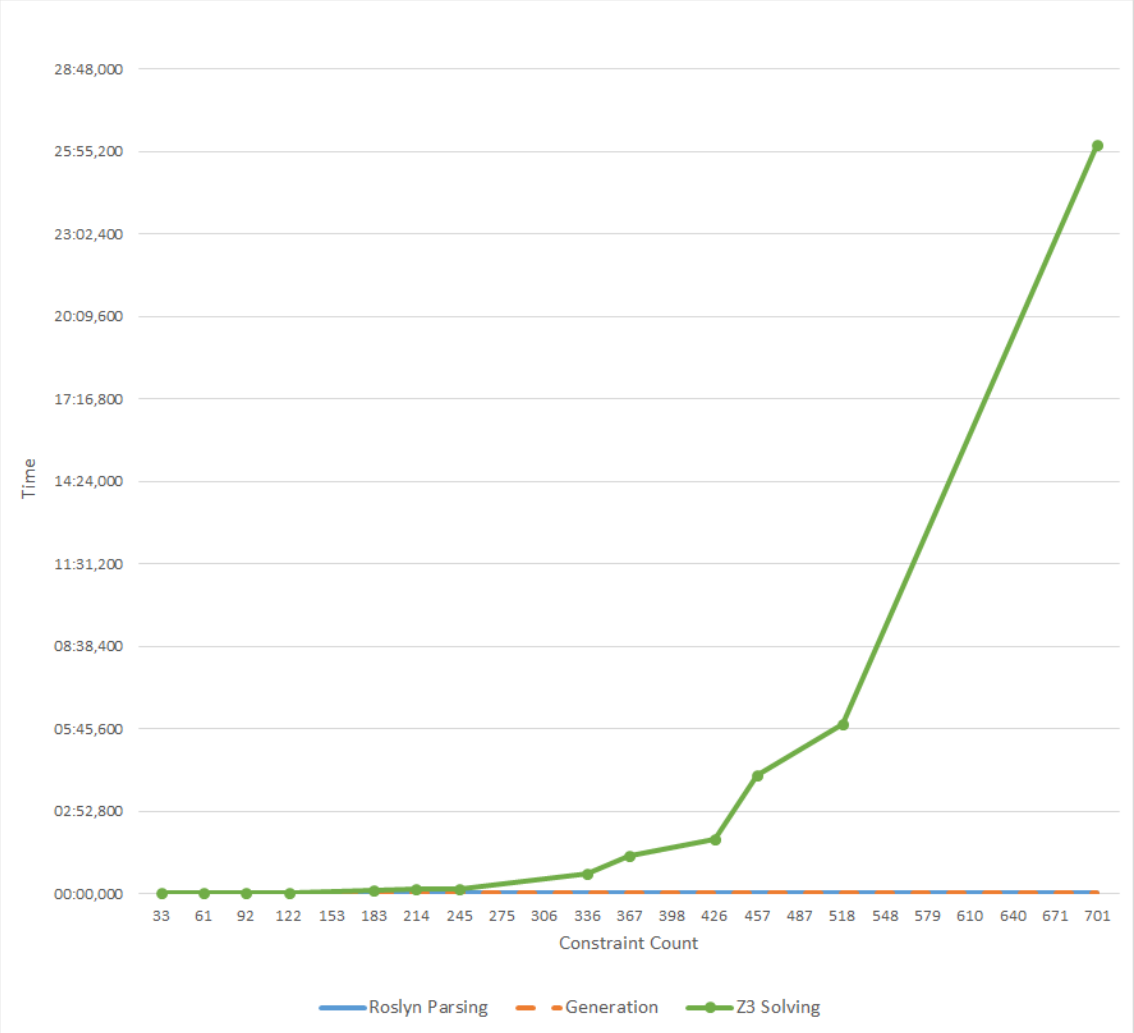
```
1  public class ScenarioClass0
2  {
3    public string PropertyString0 { get; set; }
4
5    public int PropertyInt0 { get; set; }
6
7    public Guid PropertyGuid0 { get; set; }
8
9    public ICollection<string> PropertyStringList0 { get; set; }
10
11   public ICollection<ScenarioClass0> PropertyObjList0 { get; set; }
12 }
```

During all the scenarios, the property count per class is fixed to 10 (two times the standard layout from listing 43). Scenarios for the following class counts are created and measured: 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 15, 20. During measurement, also the times for parsing the syntax trees with Roslyn are monitored. In figure 12 on the following page the results are shown.

Like in *Z3.ObjectTheorem for Entity Framework* the Z3 solving time is still exponentially, thus the cause for the observation from 5.1 are not specific EF constraints or model layouts.

[9] https://msdn.microsoft.com/en-us/library/bb126478.aspx

Figure 12: Ratio of Constraint Count to Runtime Behavior



5.3 Object Theorem Development Productivity

As shown in 5.2 LINQ and C# Expressions are not the source of slow runtime behavior. Also the current limitations from the Z3.ObjectTheorem.EF like FK constraints are limited because of the usage of the Z3 solver. This does not affect the expression tree parsing and transformation. If using C# Expression on object models, type safety and compile-time checking can be used. In addition, any knowledge about constraint generation from an object-oriented model are not needed. Thus it is very easy to use Z3.ObjectTheorem for other scenarios as Entity Framework, but the current performance limitations lead to the conclusion that another solver should be used.

6 Summary and Outlook

In this thesis it was possible to demonstrate the usage of deep fixes from [Fer15] within another environment. Moreover, it was possible to show, that C# Expressions can be used as a constraint generation language.

The evaluation leads to the conclusion that constraint-based diagnostics are not a good application scenario for the Z3 solver in terms of performance. As well the Z3 solver lacks currently a build in string type and the string theory around. In a further study it may be a try to use Z3-str2 which is an open source string theory plug-in built on the powerful Z3 SMT solver [10]. The slow runtime behavior of the Z3 Theorem Solver with the application of larger boolean theorems (large object models) leads to the question, if other solvers can handle this.

Furthermore, the actual support within Roslyn for Code Fixes is not enough for constraint-based diagnostics. The user should be able to select the needed and not needed assumptions (like in *Solution Space Explorer* at [Fer15, p. 18]), but Roslyn Code Fix cannot have a custom UI. To continue with the ideas from this thesis it would be helpful to implement a custom Visual Studio extension with a user interface which would reuse much of the Roslyn DiagnosticAnalyzer and CodeFixProvider.

In addition to this, the idea of using LINQ and C# Expressions as a constraint generation language can be viewed as a success. It does not lead to a performance bottle neck and it is possible to express all constraints with it. Also the type safety and compile-time checking can be used in contrast to the usage of the Object Constraint Language. Moreover, the ObjectTheorem simplifies the challenges for using solver which cannot understand objects and only supports functions and primitives. This type safety, compile-time checking and the possibility of automatic constraint generation leads to the opportunity to use Test Driven Development [11] during constraint specification. The next most interesting questions is, if other programming languages, like Java, also have language constructs to build a constraint generation language with the same productivity increasing attributes like above.

[10] https://github.com/z3str/Z3-str

[11] http://c2.com/cgi/wiki?TestDrivenDevelopment

A CD-ROM

Structure of folders:

- Sources: the associated source code of this thesis
- Thesis: this thesis as a PDF file

Compilation:

To compile the source code Visual Studio 2015 Update 1 is required. The complete solution can be opened with the file *Z3.ObjectTheorem.sln*. The application scenario can be opened with the file *EFScenarios.sln*.

Testing:

To run all Unit and Integration Tests, open the file *Z3.ObjectTheorem.sln* and execute *Test >Run >All Tests*.

To compile the Roslyn diagnostics and code fixes, install and run these into a second instance of Visual Studio perform the following steps:

1. Open *Z3.ObjectTheorem.sln*
2. Ensure the project *Z3.ObjectTheorem.EF.Analyzer.Vsix* is the selected StartUp project
3. Press F5 to start Debugging
4. In the newly opened Visual Studio instance open *EFScenarios.sln*
5. After solution loading the diagnostic for project *EFScenario.Bad* shows up
6. Click on the light bulb next to the DbContext class to show up the code fix
7. Click “Fix EF model” to apply the code fix

List of Listings

List of Figures

List of Tables

References

[Dav+08] David H. Akehurst et al. "C# 3.0 makes OCL redundant!" In: *Electronic Communications of the EASST* 9 (2008). URL: http://journal.ub.tu-berlin.de/eceasst/article/view/103.

[Fer15] Lehrgebiet Programmiersysteme FernUniversität in Hagen. "Dr. Deepfix or: How I learned to stop chasing error messages and love constraints." 2015.

[Mica] Microsoft. *C# Features That Support LINQ*. URL: https://msdn.microsoft.com/en-us/library/bb397909.aspx (visited on 01/03/2016).

[Micb] Microsoft. *Code First Conventions*. URL: https://msdn.microsoft.com/en-us/data/jj679962 (visited on 01/02/2016).

[Micc] Microsoft. *Code First Data Annotations*. URL: https://msdn.microsoft.com/en-us/data/jj591583 (visited on 01/02/2016).

[Micd] Microsoft. *Configuring Relationships with the Fluent API*. URL: https://msdn.microsoft.com/en-us/data/jj591620 (visited on 01/02/2016).

[Mice] Microsoft. *Configuring/Mapping Properties and Types with the Fluent API*. URL: https://msdn.microsoft.com/en-us/data/jj591617 (visited on 01/02/2016).

[Micf] Microsoft. *Custom Code First Conventions*. URL: https://msdn.microsoft.com/en-us/data/jj819164 (visited on 01/02/2016).

[Micg] Microsoft. *Enabling a Data Source for LINQ Querying*. URL: https://msdn.microsoft.com/en-us/library/bb882640.aspx (visited on 01/03/2016).

[Mich] Microsoft. *Entity Framework*. URL: https://msdn.microsoft.com/en-us/data/ef.aspx (visited on 01/02/2016).

[Mici] Microsoft. *Expression Trees (C# and Visual Basic)*. URL: https://msdn.microsoft.com/en-us/library/bb397951.aspx (visited on 01/02/2016).

[Micj] Microsoft. *How to: Implement an Expression Tree Visitor*. URL: https://msdn.microsoft.com/en-us/library/bb882521.aspx (visited on 01/02/2016).

[Mick] Microsoft. *How to: Modify Expression Trees*. URL: https://msdn.microsoft.com/en-us/library/bb546136(v=vs.90).aspx (visited on 01/03/2016).

[Micl] Microsoft. *Introduction to LINQ*. URL: https://msdn.microsoft.com/en-us/library/bb397897.aspx (visited on 01/03/2016).

[Micm] Microsoft. *LINQ (Language-Integrated Query)*. URL: https://msdn.microsoft.com/en-us/library/bb397926.aspx (visited on 01/03/2016).

[Micn] Microsoft. *NavigationPropertyNameForeignKeyDiscoveryConvention Class*. URL: https://msdn.microsoft.com/en-us/library/system.data.entity.modelconfiguration.conventions.navigationpropertynameforeignkeydiscoveryconvention.aspx (visited on 01/02/2016).

[Mico] Microsoft. *Query Syntax and Method Syntax in LINQ (C#)*. URL: https://msdn.microsoft.com/en-us/library/bb397947.aspx (visited on 01/03/2016).

[Micp] Microsoft. *System.Data.Entity.ModelConfiguration.Conventions Namespace*. URL: https://msdn.microsoft.com/en-us/library/system.data.entity.modelconfiguration.conventions.aspx (visited on 01/02/2016).

[Micq] Microsoft. *Walkthrough: Creating an IQueryable LINQ Provider*. URL: https://msdn.microsoft.com/en-us/library/bb546158(v=vs.90).aspx (visited on 01/02/2016).

[Mca] .NET Foundation Microsoft and various community contributors. *.NET Compiler Platform ("Roslyn") Analyzers*. URL: https://github.com/dotnet/roslyn-analyzers (visited on 01/03/2016).

[Mcb] .NET Foundation Microsoft and various community contributors. *.NET Compiler Platform ("Roslyn") Overview*. URL: https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview (visited on 01/03/2016).

[Mil14] Rowan Miller. *EF7 - What Does 'Code First Only' Really Mean*. Oct. 21, 2014. URL: http://blogs.msdn.com/b/adonet/archive/2014/10/21/ef7-what-does-code-first-only-really-mean.aspx (visited on 01/02/2016).

[PJ98] Jens Palsberg and C. Barry Jay. "The Essence of the Visitor Pattern." In: *COMPSAC '98 - 22nd International Computer Software and Applications Conference, August 19-21, 1998, Vienna, Austria*. 1998, pp. 9–15. DOI: 10.1109/CMPSAC.1998.716629. URL: http://www.cs.ucla.edu/~palsberg/paper/compsac98.pdf.

[Res] Microsoft Research. *Getting Started with Z3: A Guide*. URL: http://rise4fun.com/Z3/tutorial/guide (visited on 01/03/2016).

[Sme09a] Bart de Smet. *Exploring the Z3 Theorem Prover (with a bit of LINQ)*. Apr. 15, 2009. URL: http://community.bartdesmet.net/blogs/bart/archive/2009/04/15/exploring-the-z3-theorem-prover-with-a-bit-of-linq.aspx (visited on 01/02/2016).

[Sme09b] Bart de Smet. *LINQ to Z3 – Theorem Solving on Steroids – Part 0*. Apr. 19, 2009. URL: http://community.bartdesmet.net/blogs/bart/archive/2009/04/19/linq-to-z3-theorem-solving-on-steroids-part-0.aspx (visited on 01/02/2016).

[Sme09c] Bart de Smet. *LINQ to Z3 – Theorem Solving on Steroids – Part 1*. Sept. 27, 2009. URL: http://community.bartdesmet.net/blogs/bart/archive/2009/09/27/linq-to-z3-theorem-solving-on-steroids-part-1.aspx (visited on 01/02/2016).

[Ste15] Friedrich Steimann. "From well-formedness to meaning preservation: model refactoring for almost free." In: *Software and System Modeling* 14.1 (2015), pp. 307–320. DOI: 10.1007/s10270-013-0314-z. URL: http://dx.doi.org/10.1007/s10270-013-0314-z.

[US13] Bastian Ulke and Friedrich Steimann. "OCL as a Constraint Generation Language." In: *Proceedings of the MODELS 2013 OCL Workshop co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 30, 2013.* 2013, pp. 93–102. URL: http://ceur-ws.org/Vol-1092/ulke.pdf.