ORIGINAL ARTICLE

Formal Modeling and Analysis of Entity Framework Using Alloy

Bonino M.B.[1], Garis A.[2], Riesco D.[2]

[1]Universidad Tecnológica Nacional – Facultad Regional San Francisco, Argentina
[2]Universidad Nacional de San Luis, Argentina

ABSTRACT – Formal methods provide multiple benefits when applied in the software development process. For instance, they enable engineers to verify and validate models before working on their implementation, leading to earlier detection of design defects. However, most of them lack flexibility to be applied in agile software development projects. Alloy is a lightweight formal modeling language with a friendly tool that facilitates the agile approaches application. Unfortunately, its industrial adoption is hampered by the lack of methods and tools for current software development frameworks, such as Entity Framework. This platform is usually chosen by agile projects following the code-first approach that allows automatic generation of a database from domain classes coded in the C# language. We present a new method and tool for the formal specification and analysis of Entity Framework projects with Alloy. The proposal allows engineers to start the software development using Alloy for modeling, validation and verification, automatically translate Alloy specifications to C# domain classes and then generate the corresponding database with Entity Framework. We validate our approach with a real case study: an application required by a gas supplier company.

INTRODUCTION

Over the last 25 years, Formal Methods have been successfully used in the development of systems. As experts confirm in the Survey on Formal Methods (Garavel et al., 2020), the application of formal methods provides multiple benefits, such as quality, security, and easier maintenance. However, their adoption in the software industry still faces some obstacles such as the lack of efficient support tools and methods (Steffen, 2017). The daily development practice in the software industry is moving forward more rapidly than formal-methods tools and techniques, even though they are of high quality (Huisman et al. 2020). The gap becomes more pronounced in the context of agile software development projects since formal methods often follow a hardweight software engineering methodology (Kant et al., 2020).

Alloy (Jackson, 2012) is a lightweight formal language that includes a friendly model Validation and Verification (V&V) tool. Its specification language is based on first-order relational logic, with an object-oriented notation. The automatic Alloy Analyzer allows the generation of snapshots showing instances of the model as well as assertion checking of desirable properties of the system. Alloy can be relatively easily learned therefore it can be quickly introduced in agile projects (Black et al., 2009). Unfortunately, the lack of methods and tools limits Alloy's adoption in modern software development processes and platforms.

The Entity Framework (Microsoft, 2020a) platform is widely used nowadays. As an Object-Relational Mapper (ORM), it enables developers to define a database from objects, facilitating the exchange between the relational database of an application and their classes. Agile projects (Beedle et al., 2001) often choose Entity Framework to generate the database schema from the domain model classes (or data model) coded in C#, following the Code-First approach (Entity Framework Tutorial, 2019). Since Code-First frequently skips domain model analysis, the model V&V is usually outside the software development process. Therefore, defects that could be detected at this stage are possibly carried over even to the final software. Developers could tackle such a problem by using Alloy in the software development process.

We propose a new method and tool for the integration of Alloy into Entity Framework projects in order to 1) improve the quality of the domain model through Alloy, 2) automate the generation of domain classes coded in C# from an Alloy specification and 3) derive the database schema. Our aim is to not only present a method but also to implement an automatic tool to facilitate the transformation from Alloy to C#.

We validate our proposal by applying it in the development of a real system of a gas supplier company. This system was rebuilt from a version which used a traditional software development process, namely without formal methods. We then compare these two development processes in order to show the advantages of our approach.

Next section discusses RELATED WORK. Section ENTITY FRAMEWORK describes its features and basic workflow. Section INTEGRATION OF ALLOY INTO ENTITY FRAMEWORK details our proposal. Section COMPARATIVE ANALYSIS OF THE PROPOSED METHOD remarks strengths of our proposal after the comparison to another method used to develop the same case study. Last section exposes the CONCLUSION.

*CORRESPONDING AUTHOR | Bonino M.B | ✉ bonino.mb@frsfco.utn.edu.ar

RELATED WORK

There are several works in the literature that propose the transformation from Alloy to different languages for its integration into other platforms and tools. For instance, Cunha et al. propose the transformation of Alloy specifications to UML class diagrams (Cunha, Garis & Riesco, 2015), and Krings et al. introduce a translation from Alloy to B to evaluate Alloy models with AtelierB and ProB tools (Krings, Schmidt, Brings, Frappier & Leuschel, 2018).

Shriram et al. present Alchemy, a tool that compiles Alloy specifications in implementations that run in persistent databases (Krishnamurthi, Fisler, Dougherty & Yoo, 2008). Alchemy translates Alloy specifications into libraries of imperative functions. Unlike our work, Alchemy requires users to define extra mappings between the database and the code used in the application. Instead, we take advantage of the Entity Framework's features in order to reduce the code that developers must write to interact with the database.

Cunha et al. (Cunha & Pacheco, 2009) present a subset of the Alloy language and its semantic equivalence with the relational database schema, allowing translation in both directions. Unlike our proposal, the mapping from objects to the relational database is not implemented.

Some works focus on translating implementations (code or models) to Alloy specifications. Khurshid et al. (Khurshid & Marinov, 2001) present TestEra, a tool built on top of Alloy Analyzer with the objective of verifying Java implementations. They define Alloy specifications from Java code in order to use Alloy Analyzer to generate tests and verify correctness. Similarly, Vaziri et al. (Vaziri & Jackson, 2003), present a method to find errors in object-oriented code, translating Java code to Alloy specifications.

Anastasakis et al. (Indrakshi, Geri, Kyriakos & Behzad, 2010) translate UML class diagrams into Alloy specifications to detect faults in the design of UML models. Our work introduces a mapping in the opposite direction, establishing a transformation from Alloy to C#.

The combination of formal methods and C# have been studied as well. Chul-Wuk et al. (Chul-Wuk, Il-Gon & Choi, 2005) propose a tool, ACG-C# that generates implementations in C# code for security protocols verified through Casper/FDR. This approach is limited to security protocols environments, while our work is flexible enough to cover any business scope.

Barnett et al. present Spec# (Barnett, Leino & Schulte, 2004), a programming system based on the Spec# programming language, an extension of C# to build high level data abstractions. The system consists of an automatic program checker that statically verifies these specifications. Spec# is aimed at generating maintainable code of high quality, taking into account the validation of the code that is written; however, it does not provide any mechanism to perform model V&V.

ENTITY FRAMEWORK
Simulated Kalman filter

The Entity Framework (EF) is an ORM for .NET platform that differs from the classic ORM schema. A classic ORM infers that the classes have a structure similar to the database tables, but as illustrated in Figure 1, EF has a mapping layer in between that grants greater flexibility in how to get from objects to tables, and from object properties to table columns. That is, if for example, there is a C# source file with a domain class called Customer as detailed below;

```
public class Customer {
    public int Id {get;set;}
    public string FirstName {get;set;}
}
```
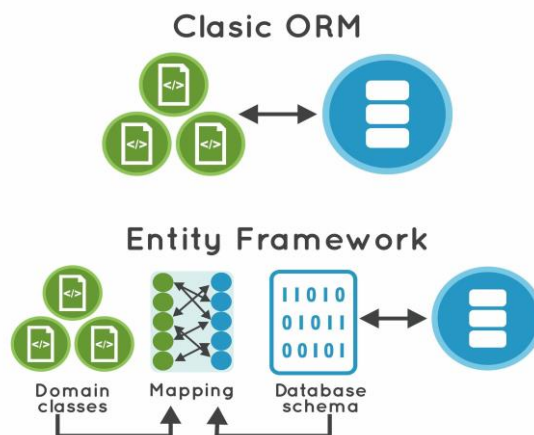

Figure 1. Comparison between a classic ORM and Entity Framework.

after mapping the database schema, it would result in a *Customer* table with the following attributes;

```
Id (PK, int, not null)
FirstName (varchar(30), not null)
```

EF introduces the Code-First approach in order to facilitate the domain design following the Domain Driven Design principle (Laribee, 2009). Therefore, developers create C# classes according to the requirements instead of specifying the database first and then the classes. Applying some conventions, the Code-First APIs generate the database on the fly conforming to the entity classes.

Conventions are specially used when classes do not match the schema of the database. They define how to infer the database schema, determine what the SQL code should be for commands and queries, and establish how to create objects from query results coming back from the database.

Figure 2 shows the EF's basic workflow. Initially the domain classes are defined, then the classes are wrapped into a model and the model mapped to the database schema using the EF's DbContext API (Entity Framework Tutorial, 2020). With the help of the database provider, EF translates queries into SQL code. It then executes the query and returns instances of the domain objects. If the application user edits, adds, or deletes an object, EF will be aware of it. Through a single command, *SaveChanges*, EF builds and executes the relevant *Insert*, *Update*, or *Delete* commands on the database.

In summary, the basic workflow includes: define the model through the domain classes, specify the context class and instruct EF through the *DbContext* class, about how these classes in the model map the database schema. The *DbContext* class is where it is defined how to use the model in the code, how the relationships are handled at runtime and how the database schema is inferred.
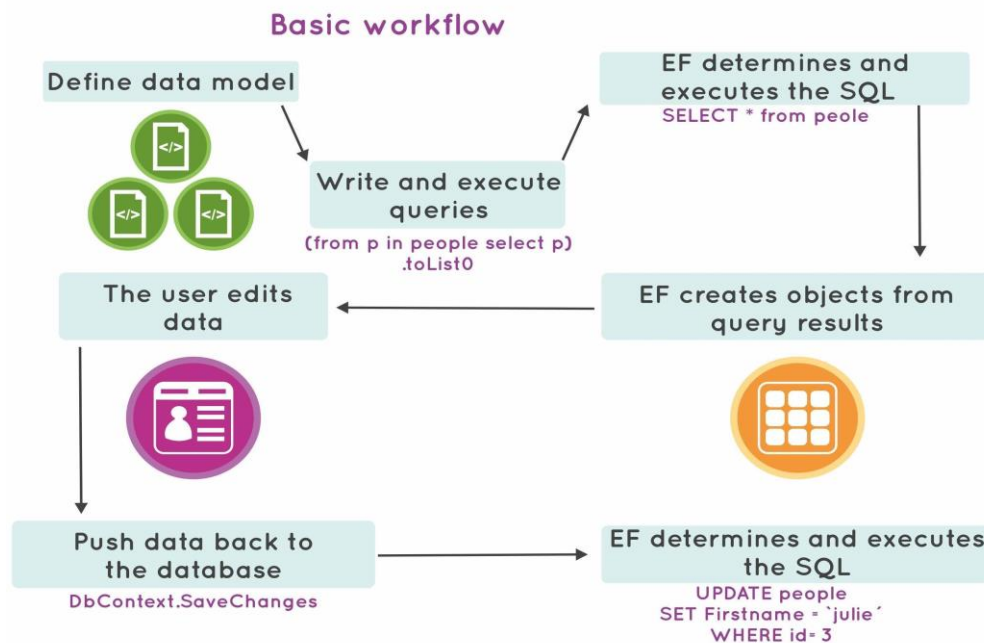

Figure 2. Entity Framework basic workflow.

INTEGRATION OF ALLOY INTO ENTITY FRAMEWORK

Figure 3 summarizes our method to integrate Alloy into EF. It includes 4 well differentiated tasks.

1. Create an Alloy model and apply V&V mechanisms to discover inconsistencies. The Alloy model should be refined until an unambiguous one is reached.
2. Automatically translate the Alloy model to C#. The C# code specifies the domain classes and the context class required by .NET environments with EF.
3. Configure the connection to the database. If necessary, the resulting C# code can be manually edited to adapt the domain classes to the database schema.
4. Migrate the database. The database is generated using the EF Migration tool.

We present a real case study to explain and validate our method. The case study is based on the analysis of the process of taking readings as part of the billing unit of a gas supplier company. An account represents a dwelling connected to the gas network and can belong exactly to a group. A group defines a segmentation of the city to organize and facilitate the taking of readings and the subsequent billing of the service. Each dwelling has one or more meters installed to determine the consumption of the service. The meter has a status that indicates the number of cubic meters released. This value increases as time passes and the user consumes the service. A reading is the status of the meter at a given time (period), which then results in the calculation of the service consumption for each dwelling. The following sections describe the tasks using this case study.
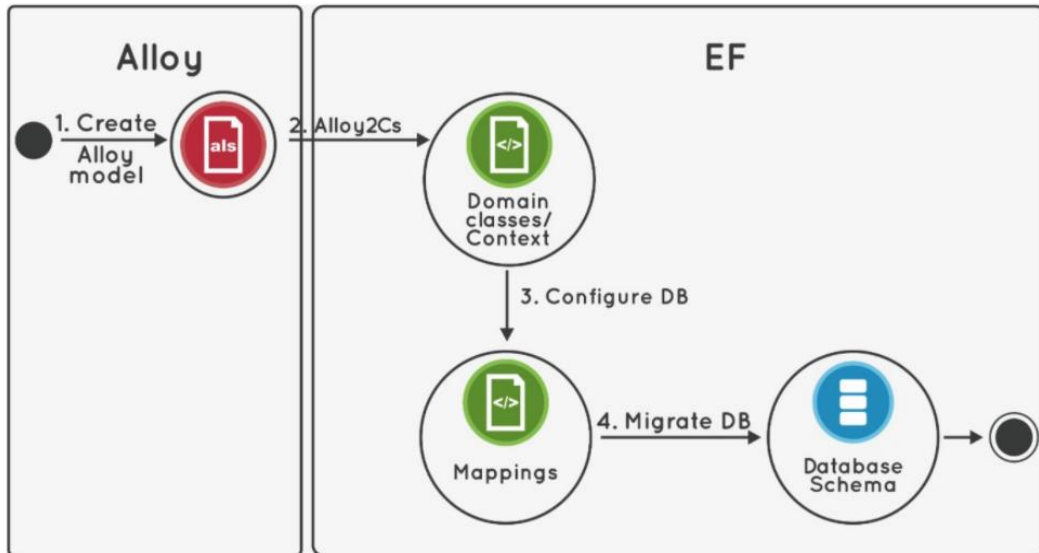
Figure 3. Workflow of the proposed method.

Creating an Alloy model and applying V&V Mechanisms

Typically, an Alloy model is created by specifying a header, signatures, invariants, predicates and functions. Additionally, assertions and commands are defined for the model V&V.

- A header, labeled by the keyword *module*, gives the module its name.
- A signature, labeled by the keyword *sig*, represents a set of atoms and may introduce some fields. A field represents a relation among signatures.
- An invariant, labeled by the keyword *fact*, establishes constraints.
- Predicates and functions, labeled by the keywords *pred* and *fun* respectively, define named constraints and expressions.
- An assertion, labeled by the keyword *assert*, allows the expression of properties that are expected to hold as consequence of specified facts.
- Commands, labeled by the keywords *run* and *check*, instruct the analyzer to find instances and counterexamples respectively.

Listing 1 illustrates an Alloy model for the case study. The *Account* signature represents a dwelling connected to the gas network. Each account belongs exactly to one *group*, represented by the field with the same name. The *readings* are specified by a field which is related to *Time*. This *Time* column indicates that the *readings* field is mutable, i.e. it can change over time.

The *Reading* signature includes three fields: *state*, *period* and *meter* to establish a reading is the state of the meter in a given period. It gives rise to the calculation of the service consumption for each dwelling.

```
module TakingReadings
sig Time {}
sig Account {
group: one Group,
readings: Reading some -> Time
}
sig Group {}
sig Reading {
state: Int,
period: Period,
meter: Meter
}
sig Meter {}
sig Period {}

fact positiveState { all r: Reading | r.state >= 0 }
fact singleAccount { all r: Reading | all t: Time | #r.~(readings.t) =
1 }
```

```
pred addReading [ a: Account, r: Reading, t,t': Time] {
r not in a.readings.t
a.readings.t' = a.readings.t + r
}
pred delReading [ a: Account, r: Reading, t,t': Time] {
r in a.readings.t
a.readings.t' = a.readings.t - r
}
run {}
```

Listing 1. Specification of the case study TakingReadings in Alloy.

The system has associated two restrictions defined with the facts *positiveState* and *singleAccount*. The first one defines that the state of the readings is always positive and the second that each reading can belong exactly to a single account. It also includes two predicates that specify operations for adding *addReading* and removing *delReading* readings, respectively; and one function *getMeterByReading* to obtain the meter related to a specific reading.

The *run* command checks the model by looking for a valid instance. Figure 4 shows an unwanted instance found when executing the aforementioned command. There are three readings -*Reading0*, *Reading1* and *Reading2*- that belong to the same account and meter, but two of them, *Reading0* and *Reading1*, are related to the same period (*Period1*). Since there can only be one reading per period per account, it means that the model is incorrect and requires new restrictions to match the rules of the domain.
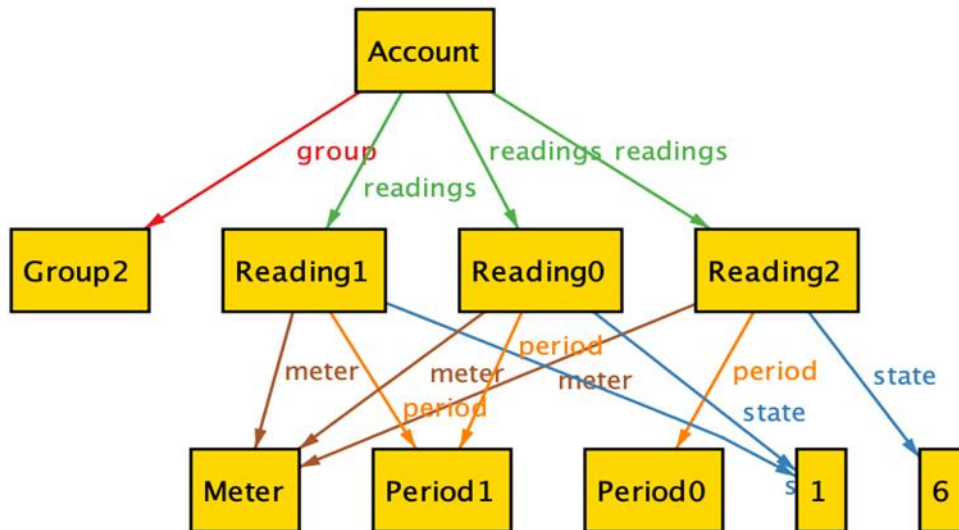
Figure 4. Instance found by the Alloy Analyze

```
fact { all m: Meter | all t:Time |
all disj r, r': Reading | (r+r') in m.~meter
=> r.period != r'.period and r.~(readings.t) = r'.~(readings.t) }
```

The *run* and *check* commands allow the model V&V to be carried out. The V&V process should be applied successively to obtain a sufficiently refined model that achieves a satisfactory representation.

Translating Alloy specifications to C# code

We developed a new tool, named Alloy2Cs, to automatically translate from Alloy to C#. The tool is implemented in Haskell as an extension of the AlloyMDA project (Cunha, Alloymda, 2016). It is available at https://github.com/nanunintan/Alloy2Cs.

We omit the specification of the correspondences between Alloy and C# elements in order to simplify the presentation. Instead, we describe the translation following the case study cited above.

Listing 2 shows the result after the translation of the case study from Alloy to C#. In general, Alloy modules are translated to C# namespaces, signatures to classes (data types), fields to properties, and predicates and functions to methods. The main difference between the methods of predicates and functions is that the first ones do not return a value and the second ones do. We translate fields to properties, but we distinguish between mutable and immutable ones.

Properties allow a class to expose a public way of getting and setting values, hiding implementation code. They can be read-write, read-only or write-only. The first one has a *get* and a *set* accessor, the second one has a *get* but no *set*, and the third one has a *set*, but no *get* accessor. The *get* and *set* accessors have different access levels and are used to return

the property value and to assign a new value, respectively. Therefore, we translate immutable fields to *read-only* properties and mutable ones to *read* and *write*.

We define the correspondences between Alloy elements and C# elements considering the following EF conventions.

Types:

A type is defined to read and write instances from/to the database. By convention, it is specified in *DbSet* properties and included in the *OnModelCreating* method. Any type that is found by recursively exploring the navigation properties is also added. For example,

```
public DbSet<Reading> Readings { get; set; }
```

specifies the *Reading* type. Subsequently it will result in a table with that name.

Properties

By default, a property is defined to read and write values from/to the database, then a public property with a getter and a setter is included. In the case of read-only properties, such as *State*, *Period* or *Meter*, only getter should be added. For example, *State* property should be translated to

```
public int State { get; }
```

however, an attribute with read-only properties will be ignored by EF and will not be translated into the database. Because EF needs to set the value of the property with the value from the database, it requires the presence of a setter accessor. Therefore, the read-only properties obtained from an Alloy model are translated to C# with *get* and *protected set*. For instance, the State property is translated to

```
public int State { get; protected set; }
```

Keys

A primary key univocally identifies each entity instance, which maps to the concept of a primary key. By convention, a property named *Id* is configured as the key of an entity. For instance,

```
public class Group { public int Id { get; set; } }
```

specifies that the *Id* property will be considered as the unique identifier of each instance of the *Group* type. Hence, it results in a primary key of the corresponding table.

Shadow properties

The most common pattern for defining a relationship is to establish navigation properties on its ends and a foreign key property in the dependent entity class. However, if no foreign key property is found in the dependent entity class, a shadow foreign key property is included. For example:

```
public class Account {
public int Id { get; set; }
public Group Group { get; set; }
public ICollection <Reading> Readings { get; set;}
}
public class Group { public int Id { get; set; } }
```

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
namespace TakingReadings {

public class DataBaseContext : DbContext {

        public DbSet<Account> Account { get; set; }
        public DbSet<Group> Group { get; set; }
        public DbSet<Meter> Meter { get; set; }
        public DbSet<Period> Period { get; set; }
        public DbSet<Reading> Reading { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {   var connectionString = @"Server=(localhost)\mssqllocaldb;
```

```
                              Database=TakingReadingsData;
Trusted_Connection=true";
                         optionsBuilder.UseSqlServer(connectionString);
        }
    }

    public class Account
    {   public int Id { get; set; }
        public Group Group { get; set; }
        public ICollection<Reading> Readings { get; set; }
        public void addReading(Reading a) {          }
        public void delReading(Reading a) {          }
    }
    public class Group
    {   public int Id { get; set; }
    }
    public class Meter
    {   public int Id { get; set; }
    }
    public class Period
    {   public int Id { get; set; }
    }
    public class Reading
    {   public int Id { get; set; }
        public int State { get; set; }
        public Period Period { get; set; }
        public Meter Meter { get; set; }
    }
}
```

Listing 2. Specification of the case study TakingReadings in C#.

results in a *GroupId* shadow property introduced to the *Account* entity, which refers to the *Id* of the entity *Group*.

Simple navigation properties

A relationship is defined with a simple navigation property. For example, the specification of *Group* in the class *Account* indicates a navigation property from *Account* to *Group*, but not the inverse.

Configuring Database

The *DbContext* class instructs EF about how the classes in the model map the database schema, how to use the model in the code, how the relationships are handled at runtime and how the database schema is inferred.
Listing 2 shows a standard definition of the connection string to the SQL Server database. Before executing the migrations and generating the real database, it is necessary to review and update the proposed parameters in the *OnConfiguring* method. For example, if it were necessary to set a connection string with a different structure, the value of the *connectionString* variable should be replaced as follows.

```
var connectionString = "data source=Server;
initial catalog=DatabaseName;
user id=DatabaseUser;
password=DatabasePassword";
```

Database Migration

The database is created using EF Migrations (Microsoft, 2016). This feature provides a way to incrementally update the database schema to keep it in sync with the application's data model, while preserving existing data.
     The initial migration, created by the *Add-Migration* command, contains the code to update the database and sync it with a set of model changes. For example,
```
     Add-Migration InitialCreate
```

     adds a new file based on the command parameter (*InitialCreate*). The file is located under the Migrations directory in the Project and contains the operations to apply the migration. After the initial migration, the *Update-Database*

command is executed in order to create the database schema. It includes the tables, relationships and elements as defined in the model.

Figure 5 shows the database corresponding to the case study *TakeReadings*. It is obtained after applying EF Migrations to the C# code generated by the Alloy2Cs tool.

The method and the tool were validated with other case studies. In particular, they were applied to Alloy models representing an address book, a file system, a genealogy and a student course. They are all available for download at the following website:
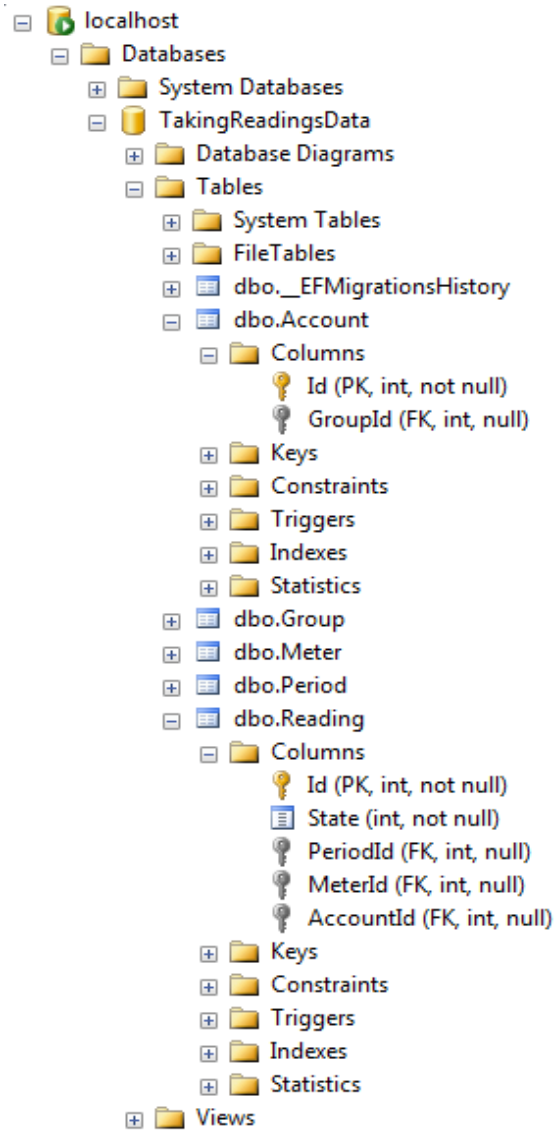
https://github.com/nanunintan/Alloy2Cs/tree/master/Alloy/Examples

Figure 5. Database generated from the C# code

COMPARATIVE ANALYSIS OF THE PROPOSED METHOD

The gas supplier case study presented in this work was rebuilt from an old version. However, this old version was created according to a different process including the following stages.

● **Modeling and Analysis**. Generation of different models to represent the aspects of interest of the system. The main focus was on obtaining UML class diagrams. No specific tool or technique was used to analyze models.
● **Data Base (DB) creation**. Manual generation of the database in SQL Server taking the previously created diagrams as a guide. A Database-First approach was taken.
● **DB interactions**. A link to the existing DB in SQL Server was defined from a Visual Studio project (Microsoft, 2017) in order to generate an Entity Data Model (EDM) with the Wizard tool. Therefore, the conceptual model from an existing database was created and the connection information to the application was added.

Table 1. Comparative table between the development processes

Stage	Old version	Our proposal
Modeling and Analysis	Class diagrams. No V&V.	Alloy model. Iterations over the model: - V&V. - Commitment and participation of the end user. - Model refinement
DB creation	Manually created in SQL Server (Database- First approach).	Automatically created using Migrations of EF (Code-First approach) Could be configured to target different DB engines.
DB interactions	EDM generation via Wizard tool. Manual generation of a class that implements access and interaction with the DB through the EDM.	Access and interaction with the DB through the EF DbContext class, automatically created by Alloy2Cs.
Mapping classes	Manual generation of the mapping classes and implementation of the business logic.	Initial mapping classes automatically created by the Alloy2Cs tool. It could require some changes in the DB configurations. Manual implementation of the business logic.

● **Mapping classes**. The classes required to represent and map information from the database into objects usable by the application, were manually created. Although Linq to Entities was used to write queries against the EDM, there was no ORM involved. This meant that all the C# code required to interact with the DB and the classes created in the previous step, had to be done manually. That is, a generic C# class was developed to create, edit, read, and delete data to and from the DB.

Table 1 compares the previous development method with the proposed in this paper. Based on this comparison, the following points are highlighted in favor of our proposal.

● Feedback obtained from early stages, as modeling, both from the development team and the end user.
  o It facilitates the understanding of the problem.
  o It facilitates and motivates the early commitment and participation of the user.
  o Immediate feedback.

  o Development team and stakeholders get an artifact that can be used to analyze, understand, and improve modeling from the beginning.
  o It allows the iteration over the model until a good enough representation that satisfies the requirements is obtained.
  o The model can be tested using instances of the Alloy Analyzer, without spending time and effort creating the necessary code that allows obtaining or simulating objects that satisfy the model.
  o Changes to be made based on the feedback have a lower cost and impact.
  o The analysis carried out through Alloy on the model specification, facilitates its evolution to an equivalent but less complex model and, therefore, clearer and easier to maintain.

● Simplification when creating the database.
  o There is no direct work with the database engine since the chosen approach is Code-First.
  o Significant reduction in time spent writing C# code.
  o Even if in the old version it had been decided to use EF as ORM and speed up the creation and interaction with the DB, it would still have been necessary to manually create the context class and models. Much of this work is done by Alloy2Cs.

● Streamlining the software development process.
  The proposed approach allows the development team to concentrate their efforts on modeling (with all the advantages already explained) and automate the next step that leads to the generation of C# code

CONCLUSION

We have presented a new method and tool for the integration of Alloy into Entity Framework. The method establishes a process for the formal specification and analysis in Alloy, the automatic transformation to C# classes (data model) through the Alloy2Cs tool, and the generation of the database within Entity Framework.

We validate our method and tool with a real system of a gas supplier company, which we also exposed as a case study to explain our approach. We compare our method to the previously used to develop the same system and we justify our method benefits.

Our proposal contributes to improving the quality of the domain model through Alloy, reducing the effort to obtain artifacts from the analysis stage with Alloy2Cs and facilitating the generation of the database schema in EF. Our work boosts the inclusion of formal methods in agile methodologies by aligning Alloy with the Code-First approach. In the future we intend to analyze its application to other software development methodologies and use different Object-Relational Mappers. We also intend to study how to include Alloy facts in C# classes, dynamic characteristics of the Alloy model, generate test cases from the Alloy specification and define the transformation from C# to Alloy.

ACKNOWLEDGEMENT

REFERENCES

[1]   Entity Framework Tutorial. (n.d.). Entity Data Model. Retrieved from What is code-first?: https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx

[2]   Black, S., Boca, P. P., Bowen, P. J., Gorman, J. & Hinchey, M. (2009). Formal Versus Agile: Survival of the Fittest?. IEEE Computer, Vol. 42, No. 9, pp. 37-45.

[3]   Beedle, M., Beck, K., Bennekum, A. v., Cockburn, A., Cunningham, W., Fowler, M., Thomas, D. (2001). Manifesto for agile software development. Retrieved from https://agilemanifesto.org/

[4]   Chul-Wuk, J., Il-Gon, K., & Choi, J.-Y. (2005). Automatic generation of the c# code for security protocols verified with casper/fdr. In Proceedings of the 9th International Conference on Advanced Information Networking and Applications, Vol. 2, pp. 507–510.

[5]   Cunha, A., Garis, A., & Riesco, D. (2015). Translating between Alloy specifications and UML class diagrams annotated with OCL. Software & Systems Modeling, Springer. Vol. 14, pp. 1-21.

[6]   Cunha, A., & Pacheco, H. (2009). Mapping between Alloy specifications and database implementations. In Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods, pp. 285-294.

[7]   Cunha, A. (2016). Alloymda. Retrieved from https://sourceforge.net/projects/alloymda/
Entity Framework Tutorial (2019). Tutorial: Get Started with Entity Framework 6 Code First using MVC 5. Retrieved from https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application

[8]   Entity   Framework   Tutorial.   (2020).   Retrieved   from   DbContext   in   Entity   Framework   6: https://www.entityframeworktutorial.net/entityframework6/dbcontext.aspx

[9]   Kant P., Hammond K., Coutts D., Chapman J., Clarke N., Corduan J. & Davies N. (2020). Flexible Formality Practical Experience with Agile Formal Methods. In Proceedings of the 21st International Symposium of Trends in Functional Programming. TFP 2020. Springer. Vol. 12222. pp. 94-120.

[10]   Krings, S., Schmidt, J., Brings, C., Frappier, M., & Leuschel, M. (2018). A Translation from Alloy to B. In Proceedings of the 6th International Conference Abstract State Machines, Alloy, B, TLA, VDM, and Z. Springer. Vol. 10817, pp. 71-86.

[11]   Krishnamurthi, S., Fisler, K., Dougherty, D. J., & Yoo, D. (2008). Alchemy: Transmuting base Alloy specifications into implementations. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp. 158-169.

[12]   Khurshid, S., & Marinov, D. (2001). Testera: A novel framework for testing java programs. In Proceedings of the 16th Annual International Conference on IEEE International Conference on Automated Software Engineering.

[13]   Garavel H., ter Beek M. & van de Pol J. (2020). The 2020 Expert Survey on FormalMethods. In Proceedings of the International Conference on Formal Methods for Industrial Critical Systems FMICS 2020: Formal Methods for Industrial Critical Systems. pp 3-69.

[14]   Huisman, M., Gurov, D., Malkis, A. (2020). Formal methods: from academia to industrial practice. A travel guide. CoRR. Vol. abs/2002.07279. https://arxiv.org/abs/2002.07279

[15]   Indrakshi, R., Geri, G., Kyriakos, A., & Behzad, B. (2010). On Challenges of Model Transformation from UML to Alloy. Software & Systems Modeling. Vol. 9, pp. 69-86.

[16]   Jackson, D. (2012). Software Abstractions: Logic, Language, and Analysis (Revised edition). MIT Press.

[17]   Laribee, D. (2009, 02). Microsoft Docs. Retrieved from Best Practice - An Introduction To Domain-Driven Design: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design

[18]   Microsoft. (2016) Microsoft Docs. Retrieved from Code First Migrations: https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/migrations/?redirectedfrom=MSDN

[19]   Microsoft. (2020a). Entity Framework Core. Retrieved from https://docs.microsoft.com/en-us/ef/core/

[20]   Microsoft. (2020b). Introduction to projects and solutions. Retrieved from https://docs.microsoft.com/en-us/visualstudio/get-started/tutorial-projects-solutions?view=vs-2019

[21]   Steffen, B. (2017). The physics of software tools: SWOT analysis and vision. Int. J. Softw. Tools Technol. Transfer. Vol. 19, pp. 1–7.

[22]   Vaziri, M., & Jackson, D. (2003). Checking properties of heap-manipulating procedures with a constraint solver. In Proceedings of 16th Annual International Conference on IEEE International Conference on Automated Software Engineering, Vol. 2619, pp. 505-520.