TOWARDS AXIOM-BASED TEST GENERATION IN .NET APPLICATIONS

Mihály Biczó and Zoltán Porkoláb

(Budapest, Hungary)

Communicated by Zoltán Horváth

Abstract. Unit testing is an important aspect of developing highly reliable and dependable applications. Although theoretically it offers the capability of testing a piece of code (typically a method) in isolation, the challenge of constructing a test set that appropriately tests the whole functionality remains open and is usually a task that programmers need to solve on an ad-hoc basis or using extreme approaches like test-driven development. This paper proposes a way how algebraic software specification can be applied to programs running on the .NET platform, how it can serve as the basis of automatic test generation and how it can replace ad-hoc testing throughout the software development process, especially during refactoring. The authors introduce the definition of a concept and an axiom, and also overview axiom-based testing in general. A mapping between the abstract definitions and the language constructs of the C# 4 programming language will be specified. Services provided by the .NET platform like attributes, reflection and call interception will be introduced and employed during implementation. It will be described how axioms differ from the contracts of the Eiffel programming language and why they are more suitable for generating test cases. The authors give the detailed description of the main components of the axiom-based test generation framework, which was implemented using .NET 4 /C# 4 and show a case study in order to demonstrate the feasibility of the solution.

1.   Introduction

During the construction of highly reliable and dependable applications, unit testing [5, 20] is an important tool in the hand of software engineers because it helps testing small pieces of code (typically methods) in isolation. However, programmers are often buried under the burden of constructing test cases that exercise the whole functionality of the code.

Mainstream software engineering has developed extreme approaches [3] like test-driven development [4, 25] in order to force developers into the habit of writing test cases even before the actual production code is constructed. In spite of its inevitable benefits, this practice does not ease the construction of a good test set, and may distract programmers' attention from the problem by requiring additional technical infrastructure elements to be used.

Would not it be nice if - with relatively small investment - one were able to automatically generate test cases that exercise a certain amount of the code being constructed?

This paper proposes a solution that gives a positive answer to the above question. The authors will focus on three important areas: 1. what kind of 'investment' is needed on the part of software engineers that enables them to automatically generate relevant test cases, 2. what is the minimal infrastructure that is needed to drive the test generation process, 3. what is the typical scenario when the proposed solution can be applied?

As for the first area, it will be shown that creating an algebraic specification [8, 9, 10, 14] will serve as a good starting point. Investigating the second area, a reference implementation will be demonstrated in the .NET 4 programming environment using the C# 4 programming language. During the investigation we will face the test data generation problem studied in [13, 19], and suggest a potential solution based on the author's previous research in the field of runtime trace generation [6, 23]. The third area - potential use cases for the framework - will be discussed briefly in the context of white-box and black-box testing and compared to code contracts used extensively by the Eiffel programming language [16, 17].

The structure of the paper is the following. Section 2 formally defines the notion of a concept, an axiom, and a model. Using these definitions clarifies the goal of axiom-based testing. Section 3 shows how these abstract notions can be mapped to program constructs that are usable in the .NET 4/C# 4 environment, and also introduces framework services like declarative descriptions (attributes), reflection and dynamic call interception. In Section 4, a reference implementation will be demonstrated for the abstract algebraic monoid type in order to show the framework in action. Section 5 summarizes the re-

sults and discusses possible future work as well as the shortcomings seen at this point. Related work in [2] served as the main motivation for the current research, although that framework was constructed for the C++ 2011 programming environment. The main contributions of this paper are: 1. it implements the concept feature in C#, 2. it introduces an axiom-based black-box testing framework.

2. Concepts, axioms, models

This section defines the notions that will be often referred to in the remaining of this paper.

2.1. The definition of a concept

According to the original concept definition presented in [2], a concept $C(p_1, p_2, \ldots, p_n) = (R; \Phi)$ consists of a set of parameters $p_1, p_2, \ldots, p_n$, a set of requirements R and a set of axioms Φ.

In this definition, the parameters can be types or operations, and a concept is an abstract notion that may have type parameters and operations over the state space spanned by the type parameters or by other fully defined types. Requirements can be predicates or other concepts. This is in accord with the concept checking principles of C++ (which are usually implemented as template meta-programs [24]). In C# 4 there are basic concept checking features built into the language itself. However, in the absence of template meta-programming, these cannot be altered without compiler modification. Compiler modification is not a feasible solution when support for mainstream development is an identified priority. Therefore, the remaining of this paper will use a simplified definition of a concept:

$$C(P; \Phi) = C(\langle p_1, p_2, \ldots, p_n \rangle; \langle \phi_1, \phi_2, \ldots, \phi_k \rangle)$$

Γ is an index set, $\mathcal{T}$ is the set of types, $\mathcal{O}$ is the set of operations, and

$$\exists \gamma \subseteq \Gamma : \forall i \in \gamma : p_i \in \mathcal{T}$$

$$\exists \delta \subseteq \Gamma : \forall j \in \delta : p_j \in \mathcal{O}$$

$$\gamma \cap \delta = \emptyset \, .$$

2.2.   The definition of an axiom

If algebraic specification is used for a data type, axioms will usually be articulated as logical comparisons over the set of operations. Operations can be divided into three categories: constructors, transformers and observers. General axiom construction studies are described in [1, 11] - a general rule being that one should construct axioms from constructor-non-constructor pairs.

2.3.   Model

A model can be thought of as a realization of the abstract notion defined by the concept. If a defined type is substituted for every type parameter in the concept, and a function for every abstract operation given in the definition of the concept, then we will gain a realization of the concept. A realization of the concept that is verified against all axioms is a model. This also implies that the set of models is potentially infinite.

As mentioned, it is expected that abstract axioms defined for a concept hold in the case of its models. This leads to the final goal of this chapter: defining the expectations about axiom-based testing.

2.4.   Axiom-based testing

As described in the previous section, the construction of a model involves type and operation substitution (and possibly, creation), and as being such, it is a process that is prone to errors.

Axiom-based testing is a means to verify that abstract axioms hold for the realization of the concept. Since axioms are defined at the abstract concept level, they must be independent of the current realization (the current model). This implies that axioms can never reflect any details that are bound to the realization process itself; therefore, they treat models as if models were black-boxes: their internal structure is irrelevant, it is only the abstract operations (and the exposed interface) that are important.

Since axioms are model independent, they operate on an infinite set of models. Using this property of axioms it is possible to establish a universal (vs. ad-hoc) black-box test bed.

The Eiffel programming language [17] has defined language elements that enable software engineers to define pre- and post-conditions for methods as well as invariants for class entities. What the present approach promotes is *black-box testing*. The Eiffel mechanism is bound to the implementation, therefore, it is usually not representation independent.

The overall goal of axiom-based testing is to verify that all the axioms hold for a new realization. This involves running axioms against an extensive set of data that is generated automatically.

3. The mapping between abstract definitions and programming language constructs

The previous section defined the abstract notions that will be employed and referred to throughout the remaining of this article. The overall goal of this section is to place these definitions into a .NET/C# context and map them to the corresponding code constructs and structures offered by this programming platform.

3.1. The realization described as a generic interface implementation

In the world of object-oriented programming [18], abstract concepts are often expressed as interfaces. In languages that support generic programming through formal type parameters, generic interfaces can be constructed as well.

```
1  public  interface  IAConcept <T>
2      where  T : class
3      {
4          T  AbstractOperation (T param );
5      }
```

Listing 1. A concept expressed in C#

C# 4 is one of these languages, therefore, it is obvious to describe concepts as generic interfaces where the formal type parameters correspond to type parameters in the abstract concept definition, and the method protocols correspond to abstract concept operations.

Therefore, a concept (without axioms) can be expressed as seen in Listing 1.

The `IAConcept` concept has one type parameter `T`, and it has one abstract operation called `AbstractOperation`. Please note that there are also concept checks prescribed for the abstract type parameter `T` (this would correspond to requirements in the original concept definition). However, this is handled by the compiler itself, so there is no need to regard them as part of the concept.

The realization of the concept is the actual interface implementation process, where specific types are substituted in the place of abstract type param-

eters, and the abstract operations are implemented using an internal representation encapsulated and hidden in the implementation class.

In other words, realizing a concept can be expressed as implementing an interface. This means that concept realization is the primary challenge of programming: it requires choosing a representation, and implementing abstract operations within the representation space.

However, this realization does not necessarily need to happen in one step. It may well be the case that the first implementation class substitutes a specific type only for type parameter `T`, and marks the `AbstractOperation` method as abstract deferring its implementation to derived classes. Whatever the implementation strategy is, in the end there must be a type that has all its formal type parameters defined and all its abstract methods implemented using some representation. On the .NET platform this may even happen in runtime, when generic types are instantiated.

For the sake of clarity, let us assume that the realization is derived in one step from the concept. In this case, the realization can look like the one shown in Listing 2.

```
1  public class ASpecificConcept : IAConcept<string>
2  {
3      //TODO: Choose representation
4
5      public string AbstractOperation(string param)
6      {
7          //TODO: Implement operation using chosen representation
8          throw new NotImplementedException();
9      }
10 }
```

Listing 2. Realization of the `IAConcept` concept interface

The implementation becomes a *model* if it is verified against all the axioms. The next section explores how axioms can be attached to the abstract concept in a non-intrusive, declarative manner.

3.2.   Axioms described as member functions

As seen previously, interfaces correspond to the notion of concepts. According to the concept definition, axioms should be attached to the concept itself. However, interfaces do not allow for method implementations or the inclusion of any implementation-dependent details, e.g. representation (data members, static data members).

One possible solution to circumvent this problem is to declare the concept methods in the interface, and implement them in derived classes. (In [2], axioms

are expressed as static member functions of a class.) Unfortunately, this means that all implementation classes of the interface must implement the axiom methods separately (which probably leads to serious code duplication [18]). Also, axioms will be exposed as public member functions, consequently, nothing would prevent clients from calling axiom methods.

Alternatively, one could implement the interface in an abstract base class realizing exclusively the axiom methods, leaving all other methods abstract. This would mostly render the code duplication problem resolved; however, in the absence of multiple inheritance, it would seriously limit the usability of the concept. Also, axiom methods would still be public. One who has performance as the primary priority on his mind may also be concerned that implementing axioms in an abstract base class would tie all future implementation to a single representation, which in turn may have a deteriorating effect on performance. However, analyzing the situation it is easy to see that implementing axioms generally does not require a representation to be chosen, and that means that the abstract base class implementing the axioms does not need to have a selected representation part.

One could say that writing axioms as member functions would have a negative effect on the logical structure of the program, and also violates object-oriented construction principles like encapsulation and data hiding. Therefore, there is a need for a less intrusive specification method that can be integrated seamlessly into new and legacy applications as well.

3.3. Axioms represented with the help of attributes

The compilation units of the .NET platform are called assemblies. Assemblies contain code in intermediate language (IL) format along with meta-data attached to certain code constructs including interfaces, classes, methods, data members or the assembly itself. Meta-data can be defined in a declarative manner using attributes. Technically, attributes are regular classes that have to inherit from the `System.Attribute` built-in framework class. For example, if we want to create an attribute by which it is possible to represent axioms, we can create the attribute class shown in Listing 3.

```
1 [AttributeUsage(AttributeTargets.Interface, AllowMultiple = true)]
2 public class AxiomAttribute : Attribute
3 {
4     public AxiomAttribute(string expression) { }
5 }
```

Listing 3. Attribute class for modeling axioms

The attribute on the attribute class itself (`AttributeUsage`) specifies the scope where it is valid to use the attribute class being defined (Attributes-

Targets.Interface), and that more than one instance of the attribute can be attached to the same target. The class itself takes a string as a constructor parameter. (There is a set of restrictions for attribute classes: a generic rule being that every 'dependency' that is necessary to instantiate the class must be known in compilation time.) Attaching the attribute to the class is easy: in square brackets one needs to specify the attribute and the actual parameters. Let us assume that we would like to express that the `AbstractOperation` method in Listing 1 is idempotent, e.g, applying it more than once to the same input does not change the result. Using the attribute class defined in Listing 3, the axiom could be attached to the concept seen in Listing 4.

```
1 [Axiom("AbstractOperation(AbstractOperation(a))==AbstractOperation
      (a)")]
2 public interface IAConcept<T>
```

Listing 4. Attaching an axiom to a concept using an attribute

This format of description does not limit the usability of the interface, does not force users to explicitly implement the axioms, and does not violate object-oriented best practices. Also, we have the advantage that the axiom is tied directly to the abstract concept (the interface).

The only question that remains open is how to retrieve attributes attached to an interface. The .NET framework provides a mechanism known as reflection, by which it is possible to search and process meta-data (attributes) attached to certain entities. If we want to get all `AxiomAttributes` attached to a given type, we can write the code fragment in Listing 5.

The `GetCustomAttributes` method can be used to get a list of all attributes of a given type. The second parameter of the function specifies if 'inherited' attributes should also be considered when creating the list. This leads us to analyzing the connection between algebraic specification and inheritance.

```
1 Type t;
2 //...
3 object[] axiomAttributes = t.GetCustomAttributes(typeof(
      AxiomAttribute), true);
```

Listing 5. Retrieving axioms attached to a concept

3.4. Axioms and inheritance

According to the Liskov substitution principle [15], functions expecting an object of a base class should behave consistently when presented with an object

of a derived class. Consequently, the derived class must satisfy the behavior specification of the base class.

In other words, the derived class must model at least the same concepts, and we should test it with the same axioms plus the axioms that are attached along the inheritance path. This means that we need the ability to track all the axioms along the inheritance path from the base (potentially interface) type to the type that we would like to test. Reflection gives us the capability to explore base types of a specific type one by one and to collect axioms attached to each of them. Using the `GetCustomAttributes` method, we only need to specify the second parameter to be true, and we gain the desired behavior.

4.   Case study

In the previous section it was shown how the abstract notions map to the program constructs offered by the .NET/C# 4 programming environment. This section will demonstrate the axiom-based testing process in detail through an example.

4.1.   The high level axiom testing process

Figure 1 depicts the process from algebraic specification until test data is generated and the corresponding axiom methods are run.
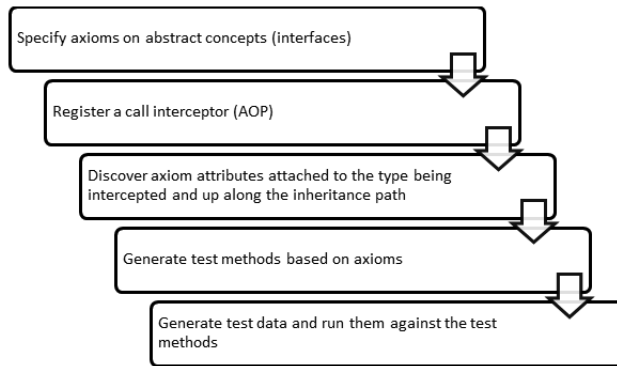

Figure 1. Process of specifying axioms and generating test data

The following section will go through this process and describe the steps in detail to clarify the concepts and to introduce the tools that are employed during implementation.

4.2. The abstract monoid type as a concept

According to the example described in [2], the algebraic monoid type was chosen. This is partly because algebraic types are usually suitable for algebraic specification, partly to emphasize differences compared to the original approach.

4.2.1. The algebraic definition of the abstract monoid

A monoid $(T, \odot)$ is a pair, where T is a set, $\odot$ is a binary operation, and the following properties hold:

$$\forall a, b \in T : a \odot b \in T$$

$$\forall a, b, c \in T : (a \odot b) \odot c = a \odot (b \odot c)$$

$$\exists e \in T : \forall a \in T : a \odot e = e \odot a = a$$

The first axiom states that the set T is closed for the operation. The second axiom requires the operation to be associative. The third axiom declares the existence of an identity element.

4.2.2. The monoid concept in C# 4

Based on the above algebraic definition, the concept definition in Listing 6 can be drawn.

```
1  [Axiom("Operation(firstArgument, Operation(secondArgument,
       thirdArgument)) == Operation(Operation(firstArgument,
       secondArgument), thirdArgument)")]
2      [Axiom("Operation(argument, Identity) == argument")]
3      [Axiom("Operation(Identity, argument) == argument")]
4      public interface IMonoid<T> : IBinaryOperation<T>,
           IIdentityProvider<T>
5      {
6      }
```

Listing 6. The monoid concept in C#

The binary operation is expressed as an interface, which is defined in Listing 7.

```
1  public  interface  IBinaryOperation <T>
2      {
3          T Operation (T firstArgument , T secondArgument );
4      }
```

Listing 7. The binary operation as an interface

The existence of the identity element is expressed as a nullary operation or property as seen from the definition of the `IIdentityProvider<T>` interface in Listing 8.

```
1  public  interface  IIdentityProvider <T>
2      {
3          T Identity { get; }
4      }
```

Listing 8. The identity element as a nullary operation - a C# property

The other axioms (associativity, identity) are expressed using the `Axiom` attribute.

4.3. Registering a call interceptor

In order to implement a mechanism for processing axioms in a non-intrusive manner (e.g., in a way that is completely transparent to the programmer), we need to register a handler that runs at every function call.

This is something that is typical in the world of aspect-oriented programming for handling cross-cutting concerns. During the author's previous research, the AOP-capabilities of the .NET framework were investigated, which produced various solutions that do not need any additional infrastructure [6, 21, 22, 23]. Also, there is a wide variety of AOP implementations for the .NET framework: Spring .NET, Castle Dynamic Proxy, PostSharp, Aspect#, etc. Besides that, most IoC containers support some kind of dynamic call interception (including the Microsoft Unity container).

For demonstration purposes, this work presents the solution using Castle Dynamic Proxy. Castle Dynamic Proxy is a library for generating lightweight .NET proxies on the fly in runtime. Proxy objects allow calls to members of an object to be intercepted without modifying the code of the class. Both classes and interfaces can be proxied; however, only virtual members can be intercepted.

Dynamic Proxy differs from the proxy implementation built into the CLR, which requires the proxied class to extend `MarshalByRefObject`. Extending `MarshalByRefObject` in order to proxy an object can be too intrusive because it does not allow the class to extend another class and it does not allow transparent proxying of classes.

Figure 2 summarizes how elements of the infrastructure using Dynamic Proxy are connected.
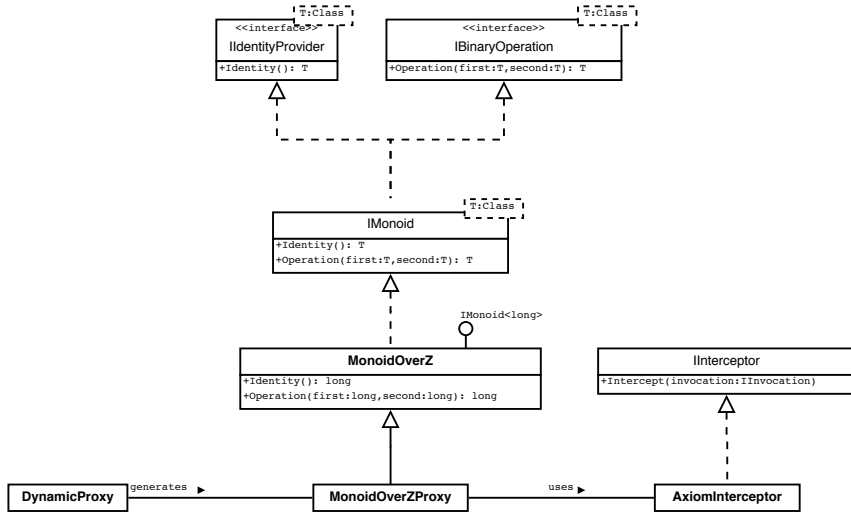

Figure 2. Static view of the infrastructure

The process is that one requests an object of an `IMonoid<T>` interface instantiated using a specific type. In the above example, a `MonoidOverZ` type was created, which is a monoid over the $\mathbb{Z}$ set. Then, the interface and the implementation type in the dependency injection container are registered.

Whenever there is a request for a `MonoidOverZ` type, the container generates a proxy (which redirects calls to the original implementation) with the exception that it provides a method hook for call interception (`AxiomInterceptor`). Whenever there is a method call to the proxy object, the `Intercept` method gets called. The `Intercept` method has one parameter of type `IInvocation`; therefore, the call context is passed to the interception method.

4.4.   Discovering axiom attributes

Using reflection, discovering axioms on the type being intercepted and also types that we inherit from is easy. All we need to do is to enumerate the interface types that the current class implements and collect attributes attached to them. This can be done using the code fragment in Listing 9.

```
1  var axioms = new List<object> { };
2               invocation.TargetType.GetInterfaces().ForEach(
3                  x => axioms.AddRange(x.GetCustomAttributes(
                       typeof(AxiomAttribute), true)));
```

Listing 9. Retrieving axioms from the concept interface

Of course, if there is a need for getting deeper in the inheritance hierarchy, it can also be done. In this example, however, it is not needed to exploit this capability.

4.5.   Generating test methods from axioms

The real challenge of the presented approach lies in the fact that attributes cannot take constructor parameters of arbitrary types. Therefore, we are forced to define axioms as string literals passed to the constructor of the `Axiom` attribute. The task is to create executable code from the string literal.

A solution was already proposed for dynamically generating lambda expressions from strings [12]. This would involve a scanner and a parser to parse the string and build an expression tree by which it is possible to create a a lambda expression. In [12] the Tiny Parser Generator was used to generate the parser and the scanner. The drawback of the approach is that it can be used only for relatively simple (LL(1)) grammars.

The parsing method can be used for pre- and post-conditions as well as invariants, but axioms may require additional parameters (not just the input parameters of a function of the class).

An alternative approach would be to generate yet another proxy dynamically in the place of the call interception that would proxy the proxy type and add the axiom methods dynamically. In this case, however, it would be necessary to create a new instance of the dynamically generated type, which may not be straightforward [7].

The generic idea is that we need to construct a scanner and a parser using which an expression tree can be built. With the help of the expression tree it is possible to emit either lambda expressions, either code that can be compiled on the fly.

For the demonstration example presented in this paper, the Tiny Parser Generator was used to generate the scanner and the parser.

4.6. Generating test data and invoking axiom method

Once the test method is created, the next task is to feed the method with appropriate test data. Various test generation techniques have emerged from the early 1970's. For demonstration purposes, a randomized test generation method over the `long` data type is employed.

The key idea is a repository of test data generators that return an appropriate test generator based on the type of the formal parameter of the axiom method. The generator repository is implemented as a static class whose `For` method retrieves a generator for the appropriate parameter type as can be seen in the code fragment in Listing 10.

The usage of the generator class happens in the place of the call interception. What we need is to iterate through the formal parameter list of the axiom method, get an appropriate `IGenerator` instance for the corresponding formal parameter, and add the generated list to the list of actual parameters as done in Listing 11.

```
 1  public static class Generator
 2  {
 3      public static IGenerator For(Type parameterType)
 4      {
 5          if (parameterType == typeof(long))
 6          {
 7              return new LongGenerator();
 8          }
 9
10          //TODO: Implement generators for other types
11
12          throw new NotImplementedException();
13      }
14  }
```

Listing 10. Test data generator repository

```
 1  var list = new List<object>();
 2                x.GetParameters().ForEach(y => InsertGeneratedValue(y,
                     list));
```

Listing 11. Inserting randomized values for formal parameters

Listing 12 shows how the `InsertGeneratedValue` method encapsulates getting the `IGenerator` instance and how to invoke the generation method.

The only piece that is missing is to invoke the axiom method using the generated values and to throw an exception if the assertion implemented by the axiom method fails.

```
1 private static void InsertGeneratedValue(ParameterInfo
      parameterInfo, List<object> list)
2 {
3     var generator = Generator.For(parameterInfo.ParameterType);
4     list.Add(generator.Generate());
5 }
```

Listing 12. Generating values for formal parameters

The invocation depends on the actual generation method chosen. In the demonstration example in Listing 13, a plain method call was used.

```
1 object invocationResult = x.Invoke(proxy, list.ToArray());
2 if (!Convert.ToBoolean(invocationResult))
3     throw new AxiomViolationException(x);
```

Listing 13. Invoking the axiom method using generated parameter values

At this point the framework is complete. Let us briefly summarize what was demonstrated in Section 4:

- A declarative, non-intrusive way to describe axioms (in [2], axioms were described as static member functions).

- A way to explore axioms attached to concepts, possibly declared higher up in the inheritance tree.

- A call interception mechanism that can be used to attach handler hooks to method calls.

- A set of methods that can be used to generate executable code from string literals.

- A way to generate test data based on the formal parameter types of the axiom method (in [2], the programmer had to supplement the generator object itself).

- A way to execute the axiom method using the generated test data in a manner that is completely transparent to the programmer.

To sum up the investigations, it can be stated that the result presented in [2] could be obtained on the .NET platform, but the approach requires less programmer interaction, and it is much less intrusive in terms of program execution and design.

5.   Summary and future work

This paper presented a framework that allows for testing axioms on the
.NET platform in a non-intrusive manner. The main contribution of the article
is that the authors ported the idea presented in [2] from C++ to .NET. An
even more important contribution is that this was done in a much less intrusive
manner: axioms can be defined on the abstract concept, and the developer does
not need to be aware of the fact that test generation and execution is being
performed in the background.

The approach presented is a typical black-box testing scenario: it cannot be
used to increase test coverage of the code or to come up with 'good' and 'sharp'
test cases. (For instance, test cases that drive execution through different
execution paths in such a way that the difference between the input data is not
larger than required to trigger a different path.)

However, black-box testing can be effectively used throughout the devel-
opment lifecycle as well as during refactoring. The research presented in this
paper is not a silver bullet. There are a lot of conceptual and technical questions
open that need to be addressed. One of them is undoubtedly the way how we
can describe axioms. The axiom described as a string literal does not provide
any design-time feedback (no intellisense support, for example). This paper
also highlighted that generating executable code from string literals passed to
attributes in constructor parameters is far from being trivial. There are more
than one possibilities, and choosing the right one for the actual case depends
on the language itself that is used to describe axioms. More complicated gram-
mars that are not LL(1) may need additional effort to fit into the proposed
framework.

Also, running axioms against the original object requires the axioms to be
side-effect free, which may not be the case for every axiom. Generating a new
object may not be straightforward [7].

Generating test data is another challenge [7]. Running axioms against ran-
domized data may be desirable during code construction. When the code is
put in production, relevant production input should be drawn and fed to the
axiom testing process. For retrieving production input data, a method was
proposed in previous research papers [6, 23]. Driving axiom based testing with
production data can protect against erroneous modifications.

A further area that needs to be thoroughly investigated is how feasible the
application of this framework is for business intensive software products. It is
probably less straightforward to create axioms for business processes than for
algebraic data types. The construction of an appropriate specification language
may be studied further.

References

[1] **Antoy, S.,** Systematic design of algebraic specifications, in: *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design*, New York, NY, USA, (1989), 278–280.

[2] **Bagge, A.H., V. David and M. Haveraaen,** Testing with axioms in C++ 2011, *Journal of Object Technology*, ETH Zurich, **10** (2010), 1–32.

[3] **Beck, K.,** Extreme programming: A humanistic discipline of software development, in: *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Springer-Verlag, (1998), 1–6.

[4] **Beck, K.,** *Test-Driven Development: By Example*, Addison-Wesley, (2002).

[5] **Beck, K. and E. Gamma,** JUnit - Java Unit testing, `http://www.junit.org` and `http://sourceforge.net/projects/junit/`, (2009).

[6] **Biczó, M., K. Pócza, I. Forgács and Z. Porkoláb,** A new concept of effective regression test generation in a C++ specific environment, *Acta Cybernetica*, **18**, (2008), 481–501.

[7] **Biczó, M.,** A state space reduction unit testing framework based on generated proxy objects, in: *Proceedings of MACS 2010*, (2010), 163–174.

[8] **Goguen, J., J. Thatcher and E. Wagner,** An initial algebra approach to the specification, correctness and implementation of abstract data types, in: Raymond Yeh (Ed.) *Current Trends in Programming Methodology*, **4**, Prentice Hall, (1978), 80–149.

[9] **Guttag, J.V. and J.J. Horning,** The algebraic specification of abstract data types, *Acta Inf.*, **10**, (1978), 27–52.

[10] **Guttag, J.V., E. Horowitz and D.R. Musser,** Abstract data types and software validation, *Commun. ACM*, **21(12)**, (1978), 1048–1064.

[11] **Guttag, J.V.,** Notes on type abstraction (version 2), *IEEE Trans.Softw. Eng.*, **6(1)**, (1980), 13–23.

[12] **Homoki, Z.,** Az Eiffel programbiztonságot támogató nyelvi eszközeinek modellezése a .NET platformon (in Hungarian), *MSc Thesis*, Eötvös Loránd University, 2009.

[13] **King, J.C.,** Symbolic execution and program testing, *Journal of the ACM*, **19(7)**, (1976), 385–394.

[14] **Liskov, B. and S. Zilles,** Specification techniques for data abstractions, in: *Proceedings of the international conference on Reliable software*, New York, NY, USA, (1975), 72–87.

[15] **Liskov, B.,** Data abstraction and hierarchy *SIGPLAN Notices*, **23(5)** (1988), 17–34.

[16] **Meyer, B.,** Applying "Design by contract", *Computer*, **25(10)**, (1992), 40–51.

[17] **Meyer, B.,** Eiffel: The language, *Prentice-Hall, Inc.*, Upper Saddle River, NJ, USA, (1992).

[18] **Meyer, B,** *Object-Oriented Software Construction*, 2nd edition, Prentice Hall Professional Technical Reference, ISBN 0-13-629155-4

[19] **Myers, G.J.,** *The Art of Software Testing*, 1st edition, John Wiley & Sons, Inc., (1979).

[20] nUnit - .NET unit testing, `http://www.nunit.org/` (Accessed: 20th June, 2011).

[21] **Pócza, K., M. Biczó and Z. Porkoláb,** Cross-language Program Slicing in the.NET Framework, in: *Conference proceedings of.NET Technologies 2005*, Plzen (Czech Republic), (2005), 141–150.

[22] **Pócza, K., M. Biczó and Z. Porkoláb,** Towards effective runtime trace generation techniques in the .NET framework, in: *Short communication papers proceedings of .NET Technologies 2006*, Plzen (Czech Republic), (2006), 9–16.

[23] **Pócza, K., M. Biczó and Z. Porkoláb,** Towards detailed trace generation using the profiler in the.NET Framework, *Annales Univ. Sci. Budapest., Sect. Comp.*, **30**, (2009), 21–40.

[24] **Siek, J. and A. Lumsdaine,** Concept checking: Binding parametric polymorphism in C++, *First Workshop on C++ Template Programming*, (2000).

[25] TestDriven .NET - test driven development in .NET, `http://www.testdriven.net/` (Accessed: 20th June, 2011).

M. Biczó and Z. Porkoláb
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
`mihaly.biczo@t-online.hu`
`gsd@elte.hu`