A Tool Generating a C# Code with Contracts of Code Contracts from a VDM++ Model with Conditions

Yuma Yamano *a1811531@sendai-nct.jp*
Department of Information Systems
National Institute of Technology, Sendai College
Sendai, 989-3128, Japan

Toshihiko Ando *tando@sendai-nct.ac.jp*
Department of Information Systems
National Institute of Technology, Sendai College
Sendai, 989-3128, Japan

Keishi Okamoto *okamoto@sendai-nct.ac.jp*
Department of Information Systems
National Institute of Technology, Sendai College
Sendai, 989-3128, Japan

Abstract

As systems rely on software, the reliability of the software is required. Formal methods are prominent ways to improve the reliability of software. Formal specification is one of the formal methods and offers a formal specification language based on mathematics and computer science. With this method, the ambiguity of the specification can be decreased, and verification can be facilitated. In development based on formal specification, specifications are formally described and then a code is generated from it. This generation is done manually in some cases, but it is done automatically by a tool in some cases. Generally, from the viewpoint of execution efficiency, etc., the generated code is modified, so it is necessary to verify whether the code meets the conditions in the specification. However, this task is manual in many cases, then it is time-consuming and error-prone. In this paper, we introduce a tool to generate a code in the programing language C# from a specification in the formal specification language VDM++. The tool also translates conditions of a specification into contracts of the library Code Contracts of #C. The above problem will be solved with this tool.

Keywords: Formal Methods, Code Generation, VDM++, C#, Code Contracts.

1. INTRODUCTION

Our society is highly dependent on software-intensive systems, for instance, automotive, medical devices, etc. Therefore, we require reliability, safety, security, etc. to the systems. However, the specification description and implementation of a software-intensive system are becoming complicated and complex as the system is becoming so. Formal methods can support these tasks to ensure a system to be reliable, safe, secure, etc. Because we can describe and verify target systems in a mathematical way with formal methods.

In this paper, we focus on formal specification while formal methods contain model checking, theorem proving, etc. A formal specification consists of a formal specification language and verification methods. We can describe a formal specification of a target system in a formal specification language so that the resulting specification has no ambiguity, can be automatically verified, and is easy to validate. Moreover, some formal specification tools can generate code in a programming language from a formal specification in a specification language. VDM (Vienna Development Method) [1] [2] is one of the formal specifications. Moreover, VDM is called a

lightweight approach [3] because it focuses on validation by testing, not on theorem proving. Many large-scale systems were developed with VDM including the contactless IC system, “FeliCa” [4].

VDM has two specification languages, VDM-SL and VDM++ [5]. With these languages, we can describe a specification of a target system, which is also called a model. The specification contains conditions that are constraints for the system besides the description of the intended behavior of the system. Thus, we can verify whether the model satisfies the conditions with tools, for instance, VDM-Tools [2] and Overture [6]. On the other hand, these tools can generate a C++/Java code from a VDM model. These tools can also translate certain conditions in a VDM model, for instance, a condition that "input of a function is less than 0". Then, we can verify the resulting source code with a condition corresponding to a condition in a VDM model. Thus, we can easily assure that the source code meets a specification. However, certain kind of conditions in a VDM model is not supported to be translated to elements in C++/Java code. Therefore, it is useful to extend the kinds of conditions in a formal specification which can be translated to conditions in code. Moreover, it is also useful to be able to generate a code in various programming languages from a formal specification in various formal specification languages.

There are related researches that develop a tool generating code in a programming language with conditions from a specification with conditions in a formal specification language. In some papers [8] [9], authors propose generation from the specification language Event-B [7] to code in a programming language. In [8], authors developed a tool that generates code in the programming language Dafny [9] from a specification in Event-B. In Dafny, we can describe conditions for a method, a function, an iterator, and a loop. These conditions are described as elements of Dafny, namely *requires, ensures, invariant*. The Dafny tool is an SMT-based verifier and runs as part of the compiler. In the paper, the tool has a restriction that descriptions in an input Event-B model must be enough refined so that data types and operators in the descriptions have corresponding counterparts in Dafny. Because some mathematical notations in Event-B are too abstract so they have no counterpart in Dafny. In [10], authors present translation rules from Event-B models to JML-annotated Java codes, implement translation as an EventB2Java tool and show two case studies of EventB2Java. On the other hand, in [11] [12], authors propose code generation from the specification language VDM-SL [5] to code in a programming language. In [11], the authors proposed a method generating C# code from a VDM-SL model, where C# [13] is an object-oriented programming language of .NET framework and supports a language extension Code Contract [14] [15] [16] to describe and verify conditions in code. They also implemented the method as a prototype for the Overture tool with which conditions in a VDM-SL model are translated to contracts in Code Contracts. In the paper, the authors state one of future works is to support the object-oriented VDM++ [5] and handle the translation of object aliasing, method overloading, multiple class inheritance, and concurrency. In [12], authors implement a tool to generate a JML-annotated Java code from a VDM-SL model.

It is important to implement a tool to generate a code with conditions of the programming language C# from a VDM++ model with conditions. Indeed, one of the authors has developed a prototype that generates code in C# with conditions from a formal specification with conditions in VDM++ [17]. However, this prototype has restrictions. The main restriction is the same as the restriction of VDM-Tools and Overture, namely the prototype cannot translate a post-condition of an operation to a description in C#. It is important to overcome this limitation because it is a fatal flaw for code verification. On the other hand, the benchmark results of [10] show that Code Contracts of C# is about 120 times faster than OpenJML [18] of Java. Therefore, C # is a promising programming language for generated source codes. A tool to generate a C# code with conditions from a VDM-SL model with conditions has been proposed in [10]. However, since the language elements of VDM-SL and VDM++ are different, it is necessary to propose a translation rule of a feature in VDM++ such as multiple class inheritance.

In this paper, we introduce a tool generating a C# code having elements of Code Contracts from a VDM++ model having pre-conditions, post-conditions, and invariants. Specifically, we introduce

additional translation rules of 1) a post-condition of an operation, 2) a condition for a type and 3) multiple inheritances of types in a VDM++ model. As a result, we extend the range of translatable descriptions in [17]. We also show the validity of the tool, in particular the correctness of the tool. More precisely, we show that our tool generates a correct code corresponding to an input VDM++ model by showing that input specifications and the generated codes pass/fail equivalent test cases.

Our tool can translate elements of VDM++ into C# elements. In particular, our tool can translate a post-condition that cannot be translated by the tool in [17]. Our tool can also translate "multiple inheritances of types" which is one of the future works of [10]. On the other hand, an automatically generated code may be manually modified to improve execution efficiency. However, manual modification is error-prone, so the modified code must be verified to meet the specifications. Our tool can support a software developer to verify whether the manually modified C# code meets a VDM++ specification by automatic generation of conditions in the C# code.

The structure of the following chapters is as follows. In Section 2, we introduce VDM++ and Code Contracts. VDM++ is an input language and Code Contracts is an extension of an output language C#. In Section 3, we introduce our tool generating a C# code having contracts from a VDM++ model having conditions. In Section 4, we show the validity of our tool. More precisely, we show that input VDM++ models and output C# models pass/fail equivalent test cases. Finally, we conclude our paper in Section 5.

2. PRELIMINARIES

In this section, we introduce a formal specification language VDM++ and a library Code Contracts of .NET Framework language. A VDM++ model is an input and a C# code, which is a .NET Framework language, is an output of our tool. There are elements of VDM++ to describe conditions for a target system. But there are no elements of C# to do that. Thus, we use elements of Code Contracts for that.

2.1. VDM++

In this subsection, we introduce the specification language VDM++ and show an example of generation from a VDM++ specification to C# code.

VDM is a lightweight formal method and has two specification languages, namely VDM-SL and VDM++. VDM++ is object-oriented while VDM-SL is not object-oriented. Since VDM is object-oriented, it has many common elements with object-oriented programming languages. For instance, VDM++ and C# have class syntax. This similarity will help us to define a translation from a VDM++ element to a C# element.

In VDM-SL and VDM++, we can describe the functions and operations of a system. Moreover, we can also describe a condition required for a system as a pre-condition, a post-condition, and an invariant. A pre-condition (a post-condition) is a condition that must be satisfied just before (respectively just after) a function or an operation is called. An invariant is a condition that must be always satisfied.

Figure 1 shows an example of a VDM++ model. In Figure 1, we define an operation *Decrement* which decrements a value of *Count* by 1. The description "*pre Count >= 1*" represents a pre-condition meaning that before executing *Decrement*, the value of *Count* must be greater or equal to 1. The description "*post Count = Count~ - 1*" represents a post-condition meaning that after executing *Decrement*, the value of *Count* is equal to the value - 1 of *Count* before the execution.

```
class Counter
values
        public static InitValue : nat = 10;
instance variables
        public Count : nat := InitValue;
operations
        public Decrement : () ==> ()
        Decrement() == (
                Count := Count - 1;
        )
        pre Count >= 1
        post Count = Count~ - 1;
end Counter
```

FIGURE 1: An Example of a VDM++ Model.

VDM-Tools and Overture can generate a C++/Java code from a VDM++ model. Figure 2 shows a part of a Java code generated by VDM-Tools from the VDM++ model of Figure 1.

```
public class Counter {
  …
  public void Decrement () throws CGException {
    if (!this.pre_Decrement().booleanValue())
      UTIL.RunTime("Precondition failure in
Decrement");
    Number rhs_4 =
Long.valueOf(Count.longValue() - 1);
    if (!UTIL.IsInteger((Object)rhs_4))
      UTIL.RunTime("Incompatible type");
    Count = TIL.NumberToLong(UTIL.clone(rhs_4));
  }
  public Boolean pre_Decrement ()
throws CGException {
    return Boolean.valueOf(Count.longValue() >= 1);
  }
}
```

FIGURE 2: The Java Code Generated by VDM-Tools.

In the VDM++ model of Figure 1, the operation *Decrement* has a pre-condition and a post-condition. However, the generated Java code of Figure 2 has only a *pre_Decrement* method, which corresponds to the pre-condition in VDM++ model, since those tools do not translate a post-condition of an operation in a VDM++ model to an element in a Java/C++ code.

2.2. Code Contracts

In this subsection, we briefly introduce Code Contracts and show an example of C# code with contracts of Code Contracts.

Code Contracts is a library to describe a contract, which is a condition for a system, in code, and is also a tool to verify contracts in .NET Framework language. With Code Contracts, we can specify contracts, namely pre-conditions, post-conditions, and invariants in .NET Framework language code, for instance, C# code.

Figure 3 shows an example of C# code with contracts of Code Contracts. In Figure 3, a method *PosSubtract* is defined. *PosSubtract* is a subtraction whose output must be greater than 0. And a

pre-condition is defined with a *Contract.Requires* method of Code Contracts in Figure 3. This *Contract.Requires* method means that the value of the first argument must be greater than the value of the second argument. Thus, the *Main* method fails because it calls the *PosSubtract* with arguments *a* and *b* such that *a* <= *b*. Indeed, static verification with Code Contracts will show us an error message in this case.

```
static void Main(string[] args) {
   int a = 2;
   PosSubtract(a, 3);
}

static int PosSubtract(int a, int b) {
   Contract.Requires(a > b);
   Contract.Ensures (Contract.Result<int>() > 0);
   return a - b;
}
```

FIGURE 3: An Example of C# Code with Code Contracts.

3. A GENERATION TOOL FROM VDM++ TO C#

In this section, we introduce our tool generating a C# code having contracts from a VDM++ model having conditions. Our tool translates a condition in a VDM++ model to a contract of Code Contracts in C# code. VDM++ and C# have many similarities since they are object-oriented languages. Moreover, a condition of VDM++ has a counterpart, which a contract, of Code Contracts. Then, a naive translation is proposed in [17] based on these similarities. However, covered elements by the translation are too restricted, we need to additionally define translation rules. We introduce the whole generation process of our tool in Subsection 3.1 and show some translation rules of our tool in Subsection 3.2.

3.1. Generation Process of Our Tool

Our tool generates a C# code (.cs files) from a VDM++ model (.vdmpp files). In this subsection, we introduce a generation process of our tool. This generation process consists of the following three steps:

(A) generating a VDM++ abstract syntax tree from a VDM++ model,
(B) translating a C#/Code Contracts abstract syntax tree from the VDM++ abstract syntax tree and
(C) generating C# code from the C# abstract syntax tree.

We adopt two existing tools to execute (A) and (C). First, we adopt VDMJ [19] at step (A). VDMJ is an open-source tool in Java and can analyze the VDM++ model syntax. Thus, we use VDMJ to get a VDM++ abstract syntax tree from a VDM++ model. Second, we adopt the .NET compiler platform Roslyn [20] and MSBuild for (C). Roslyn is API for .NET compiler functions. With Roslyn and MSBuild, we can generate C# code from a C# abstract syntax tree. The rest is a tool to execute (B).

Based on [17], we define a translation rule from an element of VDM++ to an element of C# and Code Contracts for (B). However, the translation rule in [17] is too restricted, so we must additionally define translation rules. In particular, we additionally define a correspondence between an element of VDM++ and an element of C# and Code Contracts and show a part of this correspondence in Table 1. We will show details of some translation rules in the next subsection. Then, we implement a function that recursively translates from a VDM++ abstract syntax tree to a C# abstract syntax tree.

VDM++	C#
class	class / interface
function definitions	method
operation definitions	method
value definitions	member variable
type definitions	inner class
instance variable definitions	member variable
pre-conditions	*Contract.Requires* method
post-conditions	*Contract.Ensures* method
invariants	*Contract.Invariant* method

TABLE 1: A Correspondence between Elements of VDM++ and Elements of C# and Code Contracts.

3.2. Translation Rules of Our Tool

In this subsection, we show some translation rules for our tool.

VDM++ and C# have many similarities since they are object-oriented languages, and naive translation rules are shown in [17]. However, there are still some gaps between them. We fill these gaps. We show translation rules of 1) a post-condition of an operation, 2) a condition for a type and 3) multiple inheritances of types in a VDM++ model.

1) We define a translation rule from a post-condition of an operation in a VDM++ model to a *Contract.Ensures* method of Code Contracts in C# code. Because, in Java, after calling a method having a variable *x*, we cannot refer to the value that is assigned to *x* before the method is called. In contrast, we can refer it to C# with Code Contracts.

Figure 5 shows the C# code generated from the VDM++ model of Figure 1 by our tool. While the generated Java code of Figure 2 does not contain a contract corresponding to the post-condition in the VDM++ model of Figure 1, the generated C# code of Figure 5 contains a contract (a *Contract.Ensures* method) corresponding to the same post-condition.

```
public class Counter {
   public static readonly uint InitValue = 10;
   public uint Count = InitValue;

   public void Decrement() {
      Contract.Requires(Count >= 1);
      Contract.Ensures(Contract.Equals(Count, Contract.OldValue(Count) - 1));
      Count = Count - 1;
   }
}
```

FIGURE 5: The Generated C# Code from the VDM++ Model of Figure 1.

2) We define a translation rule from a VDM++ type definition to a C# inner class. Because a VDM++ type definition often has invariants and an invariant of a VDM++ type definition can be translated to a C# method in a C# inner class.

Figure 6 shows a VDM++ model that contains a type definition, and Figure 7 shows the generated C# code from the VDM++ model of Figure 6 by our tool. In Figure 6, the VDM++ class *TypeTest* contains the type definition "*public Pin = int"* with an invariant "*inv n == n < 30*". In this case, our tool generates a C# class *TypeTest* and a C# inner class *Pin* in the C# class *TypeTest* such that the inner class *Pin* contains an *ObjectInvariant* method corresponding to the invariant in the VDM++ type *Pin*.

```
class TypeTest

types
public Pin = int
inv n == n < 30;

end TypeTest
```

FIGURE 6: A VDM++ Model with a Type Definition.

```
public class TypeTest {
public class Pin : VDMUtil.IType, ICloneable{
        public Pin(long value) {
            Value = value;
        }

        public long Value { set; get; }

        override public bool Equals(object ob) {
            if (null == ob) {
                return false;
            }

            if (ob.GetType() != this.GetType())
                return false;
            if (this.Value != (ob as Pin).Value)
                return false;
            return true;
        }

        override public int GetHashCode() {
            return this.Value.GetHashCode();
        }

        [ContractInvariantMethod]
        private void ObjectInvariant() {
            Contract.Invariant(Value < 30);
        }
    }
}
```

FIGURE 7: The generated C# code from the VDM++ model of Figure 6.

3) We define a translation rule from a VDM++ class to a C# class or a C# interface. In VDM++, a class can inherit multiple classes. In contrast, in C#, a class cannot inherit multiple classes while a class can inherit multiple interfaces. However, certain VDM++ classes cannot be translated into a C# interface. Thus, referring to the result of [21], we define the following four conditions of a VDM++ class to determine whether a VDM++ class can be translated into a C# interface:

- all functions and operations defined in the class having a body are subclass responsibility,
- all functions and operations in the class are public,
- there are no instance variable definition and value definition in the class and
- all superclasses of the class are translated to C# interfaces.

When our tool detects a VDM++ class satisfying these conditions, it allows a user to choose a C# interface and a C# class as a result of the translation of the VDM++ class. Moreover, if a VDM++ class contains a function or an operation that is designated as subclass responsibility, the VDM++ class is translated to a C# abstract class.

Figure 8 shows a VDM++ class satisfying the above four conditions, and Figure 9 shows the translated C# interface while Figure 10 shows the translated C# class.

```
class ITest

operations
public op : nat ==> nat
op(a) == is subclass responsibility;

functions
max: int * int -> int
max(x, y) == is subclass responsibility;

end ITest
```

FIGURE 8: A VDM++ Class Satisfying the Four Conditions.

```
public interface ITest {

   [Pure]
   long max(long x, long y);

   ulong Test(ulong a);
}
```

FIGURE 9: The C# Interface Translated from the VDM++ Class of Figure 8.

```
public abstract class ITest {

   [Pure]
   public abstract long max(long x, long y);

   public abstract ulong Test(ulong a);
}
```

FIGURE 10: The C# Class Translated from the VDM++ Class of Figure 8.

4. VALIDITY OF OUR TOOL

In this section, we show the validity of our tool, in particular, the correctness of translation rules. To show the correctness of the translation, it is enough to show that an input VDM++ model and the generated C# code are equivalent. More precisely, we show that a VDM++ model and the generated C# code pass/fail equivalent test cases. Moreover, we show the capability of translation of various language elements. In actual development, we make abstract specifications and then refine it to concrete specifications. Thus, we adopt an abstract (naïve) specification S1 that is easy for humans to understand and a concrete (efficient) specification S2 that is difficult for

humans to understand as input models. We show that the specification S1 (S2) and the generated code C1 (respectively C2) pass/fail equivalent test cases. Moreover, S1 and S2 are proved to be logically equivalent in [22], we show that S1, S2, C1, C2 pass/fail equivalent test cases. We conduct testing for a VDM++ model with VDM-Tools, and for the generated C# code with Visual Studio.

We adopt example C codes in [22] (Figure 11 and Figure 12) as sources of input VDM++ models. These C codes are codes to identify the position of the first 1 bit in a word. They are proved to be equivalent, which means that they generate the same output for the same input. The code in Figure 11 is a naive implementation and the code in Figure 12 is an efficient implementation.

```
uint32_t ffs_ref(uint32_t word) {
    int i = 0;

    if(!word)
        return 0;

    for(int cnt = 0; cnt < 32; cnt++)
        if(((1 << i++) & word) != 0)
            return i;

    return 0; // notreached
}
```

FIGURE 11: A Source of an Input VDM++ Model (a Naïve Implementation).

```
uint32_t ffs_imp(uint32_t i) {
    char n = 1;
    if (!(i & 0xffff)) { n += 16; i >>= 16; }
    if (!(i & 0x00ff)) { n += 8;  i >>= 8; }
    if (!(i & 0x000f)) { n += 4;  i >>= 4; }
    if (!(i & 0x0003)) { n += 2;  i >>= 2; }

    return (i) ? (n+((i+1) & 0x01)) : 0;
}
```

FIGURE 12: A Source of an Input VDM++ Model (an Efficient Implementation).

We construct a VDM++ model in Figure 13 (Figure 14) of a C code in Figure 11 (respectively Figure 12). Then we generate the C# code in Figure 15 (Figure 16) from the model in Figure 13 (respectively Figure 14) with our tool.

```
public ffs_ref : Byte ==> int
ffs_ref(k) == (
tmp := 0;
if k.isZero() then return 0;
for cnt = 0 to 31 by 1 do (
  one := new Byte(false, false, false, true);
  one.shiftLeft(tmp);
  tmp := tmp + 1;
  if not one.AND(k).isZero()
  then return tmp;
);
return 0;
);
```

FIGURE 13: A VDM++ Model of the Code of Figure 11.

```
public ffs_imp : Byte ==> int
ffs_imp(i) == (
tmp := 1;

if i.AND(new Byte(true, true, true, true)).isZero() then (
tmp := tmp + 4;
i.shiftRight(4);
);

if  i.AND(new Byte(false, false, true, true)).isZero() then (
tmp := tmp + 2;
i.shiftRight(2);
);

if i.isZero() then return 0;
if i.b1 then return tmp;

return tmp + 1;
);
```

FIGURE 14: A VDM++ Model of the Code of Figure 12.

```
public long ffs_ref(Byte k) {
    tmp = 0;

    if (k.isZero())
        return 0;

    for (var cnt = 0; cnt <= 31; cnt += 1) {
        one = new Byte(false, false, false, true);

        one.shiftLeft(tmp);
        tmp = tmp + 1;

        if (!one.AND(k).isZero())
            return tmp;
    }

    return 0;
}
```

FIGURE 15: The Generated C# Code from the VDM++ Model of Figure 13.

```
public long ffs_imp(Byte i) {
  tmp = 1;

  if (i.AND(new Byte(true, true, true, true)).isZero()) {
      tmp = tmp + 4;
      i.shiftRight(4);
  }

  if(i.AND(new Byte(false,false,true,true)).isZero()) {
      tmp = tmp + 2;
      i.shiftRight(2);
  }
```

```
  if (i.isZero())
     return 0;
  if (i.b1)
     return tmp;

  return tmp + 1;
}
```

FIGURE 16: The Generated C# Code from the VDM++ Model of Figure 14.

We write test cases for VDM++ models and translated them to equivalent test cases for C# codes. Then, we execute the equivalent test cases to VDM++ models and C# codes. Thus, we find that all the models and codes pass/fail the equivalent test cases. This result shows that the translation rules in our tool will be valid.

5. CONCLUSION AND FUTURE WORK

In this paper, we introduce a generation tool from VDM++ to C# with Code Contracts. In particular, we additionally define translation rules from an element of a VDM++ model with conditions to an element of C# code with contracts of Code Contracts. Moreover, we show the validity of our tool by showing that input VDM++ models and the generated C# codes pass/fail equivalent test cases.

Our generation tool supports software developers in the following two ways. First, our generation tool allows C# as a new choice of a target programing language of generation from VDM++. While, with existing tools [2] [6], it is possible to generate from VDM++ only to Java and C++. Second, our tool can translate pre-conditions, post-conditions, and invariants in a specification (a VDM++ model) into contracts of Code Contracts in C# code. While the existing tool [17] does not support to translate a post-condition of an operation. Thus, since conditions in a specification are translated into a contract in the generated program, it is easy to check whether the program (the C# code) meets conditions in a specification (a VDM++ model).

We have two future works. First, we will increase the number of translatable VDM++ elements, including the record type of VDM++. It will enhance the capability of the translation of language elements of VDM++. Second, we will define translation rules from conditions of a VDM++ model to properties of property-based testing [23] as the conditions will also contribute to defining properties of property-based testing.

6. REFERENCES

[1] J. Fitzgerald, P.G. Larsen. Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge: Cambridge University Press, 2009, pp. 1-228.

[2] Kyushu University. “FM VDM”, Internet: fmvdm.org/, [Aug. 28, 2020].

[3] S. Agerholm, P.G. Larsen. “A Lightweight Approach to Formal Methods,” in Applied Formal Methods — FM-Trends 98. FM-Trends 1998. Lecture Notes in Computer Science, vol. 1641. 1999, pp. 168-183.

[4] T. Kurita, F. Ishikawa and K. Araki. “Practices for Formal Models as Documents: Evolution of VDM Application to Mobile FeliCa IC Chip Firmware,” in Formal Methods - 20th International Symposium, 2015, pp. 593-596.

[5] P.G. Larsen, K. Lausdahl, N. Battle, J. Fitzgerald, S. Wolff, S. Sahara, M. Verhoef, P.W.V. TranJørgensen, T. Oda, P. Chisholm. “VDM-10 Language Manual.” Internet: overturetool.org/documentation/manuals.html [Oct. 5, 2020].

[6] “Overture Tool.” Internet: overturetool.org, [Aug. 28, 2020].

[7] J.R. Abrial. Modeling in Event-B: system and software engineering. Cambridge: Cambridge University Press, 2010, pp. 1-586.

[8] M. Dalvandi, M. Butler and A. Rezazadeh. “From Event-B Models to Dafny Code Contracts,” in 6th IPM International Conference on Fundamentals of Software Engineering (FSEN 2015), 2015, pp. 308-315.

[9] K.R.M. Leino, “Dafny: An automatic program verifier for functional correctness.” in Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2010. Lecture Notes in Computer Science, vol. 6355. E.M. Clarke, A. Voronkov, Ed. Berlin, Heidelberg: Springer, 2010, pp. 348-370.

[10] V. Rivera, N. Catano, T. Wahls and C. Rueda. “Code generation for Event-B.” International Journal on Software Tools for Technology Transfer, vol.19, pp.31-52, Feb. 2017.

[11] S. Diswal, P.W.V. Tran-Jørgensen and P.G. Larsen. “Automated Generation of C# and .NET Code Contracts from VDM-SL Models,” in THE 14TH OVERTURE WORKSHOP, 2016, pp. 32-47.

[12] P.W.V. Tran-Jørgensen, P.G. Larsen and G.T. Leavens. “Automated translation of VDM to JML-annotated Java.” International Journal on Software Tools for Technology Transfer, volume 20, pp. 211–235, Apr. 2018.

[13] Microsoft. “C# reference.” Internet: docs.microsoft.com/en-us/dotnet/csharp/language-reference/, Feb. 14, 2017 [Oct. 5, 2020].

[14] Microsoft Research. “Code Contracts.” Internet: research.microsoft.com/en-us/projects/contracts/, [Aug. 28, 2020].

[15] M. Barnett, K.R.M. Leino and W. Schulte. “The Spec# Programming System: An Overview,” in Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. CASSIS 2004. Lecture Notes in Computer Science, vol. 3362. G. Barthe, L. Burdy, M. Huisman, JL. Lanet JL and T. Muntean, Ed. Berlin, Heidelberg: Springer, 2005, pp. 49-69.

[16] D. Strauss. (2016, Mar. 18). C# Code Contracts Succinctly. [On-line]. Available: www.syncfusion.com/ebooks/csharpcontracts [Oct. 5, 2020].

[17] Y. Chisaka and K. Okamoto. “A prototype of a translation tool from VDM++ to C#,” in Proceedings of the 78th National Convention of IPSJ. 2016, pp.363-364.

[18] D.R. Cok. “OpenJML: JML for Java 7 by Extending OpenJDK,” in NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 472–479. M. Bobaru, K. Havelund, G.J. Holzmann and R. Joshi, Ed. Berlin Heidelberg: Springer, 2011, pp. 472-479.

[19] N. Battle. “VDMJ.” Internet: github.com/nickbattle/vdmj, [Aug. 28, 2020].

[20] GitHub. “dotnet/Roslyn.” Internet: github.com/dotnet/roslyn, [Aug. 28, 2020].

[21] Kyushu University. “VDMTools User Manual.” Internet: github.com/vdmtools/vdmtools/raw/stable/doc/user-man/usermanpp_a4E.pdf, [Oct. 5, 2020]

[22] Galois inc. “SAW – tutorial.” Internet: saw.galois.com/tutorial.html, [Aug. 28, 2020].

[23] K. Claessen and J. Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, 2000, pp. 268–279.