Automated Generation of C# and .NET Code Contracts from VDM-SL Models

Steffen P. Diswal, Peter W. V. Tran-Jørgensen and Peter Gorm Larsen

Department of Engineering, Aarhus University, Finlandsgade 22, 8200 Aarhus N, Denmark

Abstract. Automatic code generation gives software engineers a convenient way to realise a VDM-SL specification in a programming language in order to achieve interoperability with standard libraries and external software modules. This paper presents a set of rules for translating VDM-SL to C#. The rules have been implemented in a proof-of-concept transcompiler as an extension to the Overture tool. The performances of the .NET Code Contracts library and the OpenJML tool are measured and compared, revealing a significant computational overhead in the latter.

Keywords: VDM-SL, C#, JML, .NET Code Contracts, code generation, Design-by-Contract

1 Introduction

Being a formal modelling language, VDM-SL differs significantly from low-level programming languages by focusing on high-level abstractions rather than pointers and details of the underlying hardware platform [8]. Combined with the Design-by-Contract (DbC) elements in VDM-SL, which enable automatic software verification, these language traits support the early analysis of software systems. However, eventually a VDM-SL model needs to be realised in a programming language. It is therefore of interest to investigate to what extent it is possible to automate the efforts needed to translate a VDM-SL specification into contract-aware code.

This paper is based on the master's thesis by the first author [7], which presents a translation of VDM-SL to C# programs with .NET Code Contracts. The translation covers a substantial part of the executable subset of VDM-SL, and the core of that is presented in this paper.

In [19], Jørgensen et al. present an approach to translating VDM-SL specifications to Java Modelling Language (JML) [4] annotated Java programs. Their translation is presented as rules that are implemented in Overture's Java code generator in order to make the approach fully automated. Similar to our work, the JML generator is developed using Overture's code generation platform [9]. However, our work is different in that we target different implementation technologies, i.e. C# and .NET Code Contracts. In particular, there are differences between JML and .NET Code Contracts which do not allow us to directly use the rules presented in [19]. The reason is that the DbC elements supported by JML and .NET Code Contracts differ. For example, in .NET Code Contracts, invariants are only checked at the end of public methods, whereas in JML, they

must hold in the pre- and post-states of every method, unless the method is declared as a JML helper. Furthermore, JML uses ghost variables to specify abstract state of a class – this construct does not exist in .NET Code Contracts.

Other noteworthy attempts have been made to represent VDM-SL's DbC constructs using contract-aware code. In [21], Visser et. al. present an approach that uses Haskell's monad construct to offer several modes of runtime evaluation. To name a few examples, the *error* mode raises exceptions when contracts are violated, whereas the *free fall* mode performs no checking of contracts at all. Free fall mode may be desired for production code when system performance is of high importance. One advantage of using monads to represent contracts is that function definitions do not need to be augmented with extra contract checks as this is being handled by monads.

In terms of modelling technologies other than VDM, attempts have been made to translate Event-B models [1] into contract-aware code. In [16], Rivera et. al. present the EventB2Java code generator, which translates both abstract and refinement Event-B models into JML annotated Java programs. Similar to our work, EventB2Java is fully automated and does not require any user intervention. In [6], Dalvandi et al. present an approach to the translation of Event-B specifications into Dafny code contracts [11]. While their code generator leaves the implementation of the Dafny classes to the user, our code generator also translates the explicit parts of function and operations (the bodies) into code.

The remaining part of this paper is structured as follows: Section 2 introduces the parts of C# and the .NET Code Contracts library necessary to understand the translation. Section 3 presents an extract of the rules used to generate C#/.NET Code Contracts from VDM-SL. Section 4 compares the performance of the .NET Code Contracts library to the OpenJML runtime-assertion checker, and finally, Section 5 concludes this paper and outlines future work.

2 C# and .NET Code Contracts

.NET is a software platform invented by Microsoft in 2001[1]. The core of .NET is the ISO- and Ecma-standardised Common Language Infrastructure (CLI) specification [15] (see Fig. 1). It defines a platform-independent assembly language, Common Intermediate Language (CIL), as well as the architecture of the corresponding runtime environment, Virtual Execution System (VES). Additionally, it defines a set of standard libraries that can be used by all CLI-compliant programs. The primary implementation of the CLI is the *.NET Framework*, which targets the Windows operating system family. In this context, the Common Language Runtime (CLR) is a virtual machine that implements VES.

Alongside the development of .NET, a development team at Microsoft lead by Anders Hejlsberg conceived the C# programming language [12]. Primarily influenced by C++ and Java, it is an object-oriented language that targets the .NET platform by compiling to CIL. Its syntax resembles that of Java to a large extent.

[1] See https://dot.net.

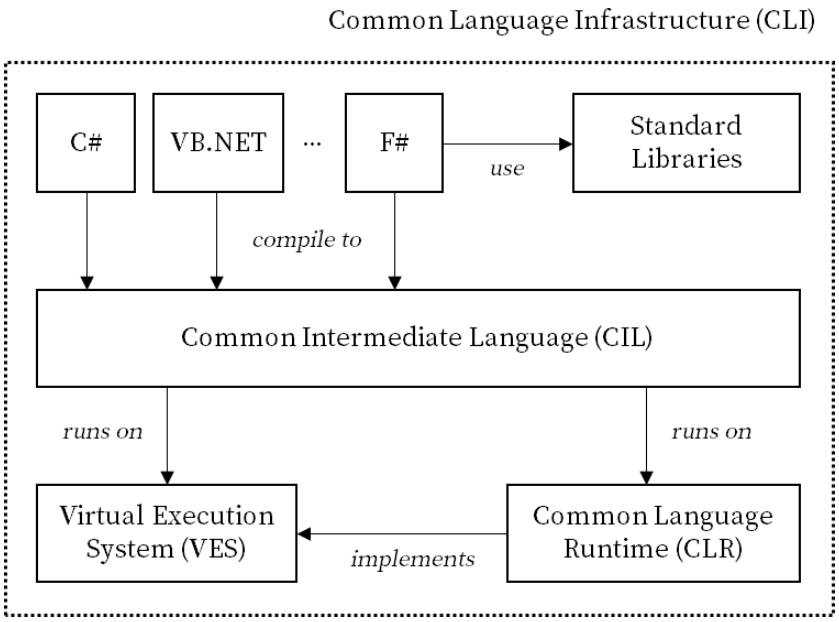

Fig. 1. The architecture of the Common Language Infrastructure in .NET.

3 Example Rules for Translation

3.1 Design-by-Contract

C# supports DbC through the .NET Code Contracts library [18, 17], which is a spin-off from the Spec# research project [2]. The .NET Code Contracts library resides under the `System.Diagnostics.Contracts` namespace and supports preconditions, postconditions, invariants as well as explicit assertions. When contracts are violated the library throws an exception. Contract enforcement is enabled by defining a symbol named `CONTRACTS_FULL` via the **`#define`** directive in C#.

Pre- and postconditions In .NET Code Contracts, preconditions are specified by calling the `Contract.Requires` method at the beginning of the method being guarded. Similarly, postconditions are specified by calling `Contract.Ensures`. They take a logical predicate as input and throw an instance of `ContractException` if the predicate evaluates to false when the C# method is entered and exited, respectively. The predicates must be referentially transparent and may only call methods tagged with the `PureAttribute` type – *pure methods* in .NET terminology – and read variables. Because VDM-SL treats the logical predicates in pre- and postconditions as self-contained Boolean functions [10], they are equivalent to pure **`bool`** methods in C# and may be called within contract predicates.

Within the predicate of a postcondition, it is also possible to refer to the guarded method's return value by calling `Contract.Result<TReturn>`, where `TReturn`

is the method's return type. Furthermore it is also possible to refer to the method's pre-state by calling `Contract.OldValue`, which takes an expression as input and returns the value of this expression at the time of entry into the method.

Translating preconditions for functions

Let f : X * Y * Z -> R be a VDM-SL function guarded by a precondition p, let e_{p} be the logical predicate of p, and let f_{pre} : X * Y * Z -> **bool** be the self-contained function for p in VDM-SL. Then f and f_{pre} become pure methods f' and f'_{pre} in C#. f' calls `Contract.Requires(`f'_{pre}`(`x`, `y`, `z`))` for parameters x, y and z. f'_{pre} evaluates and returns e_{p}.

Translating postconditions for functions

Let f : X * Y * Z -> R be a VDM-SL function guarded by a postcondition q, let e_{q} be the logical predicate of q, and let f_{post} : X * Y * Z * R -> **bool** be the self-contained function for q in VDM-SL. Then f and f_{post} become pure methods f' and f'_{post} in C#. f' calls `Contract.Ensures(`f'_{post}`(`x`, `y`, `z`, Contract.Result<`R'`>()))` for parameters x, y and z and return type R'. f'_{post} evaluates and returns e_{q}.

The VDM-SL specification of an Automated Teller Machine (ATM) has been used to demonstrate Overture's JML translator [19]. The same example is used here with focus on DbC elements to compare the translations to JML and C#. A VDM-SL operation such as:

```
operations
    AddCard: Card ==> ()
    AddCard(c) == validCards := validCards union {c}
    pre c not in set validCards
    post c in set validCards;
```

is in a C#/.NET context translated to:

```
public static void AddCard(Card c) {
    Contract.Requires(c != null);
    Contract.Requires(PreAddCard(c, State));
    Contract.Ensures(
        PostAddCard(c, Contract.OldValue(State), State));
    State.ValidCards.Add(c);
}

[Pure]
public static bool PreAddCard(Card c, St st) {
    Contract.Requires(c != null);
    Contract.Requires(st != null);
    return !st.ValidCards.Contains(c);
```

```
}

[Pure]
public static bool PostAddCard(Card c, St oldSt, St st) {
    Contract.Requires(c != null);
    Contract.Requires(oldSt != null);
    Contract.Requires(st != null);
    return st.ValidCards.Contains(c);
}
```

Invariants Invariants in .NET Code Contracts are specified by calling `Contract.Invariant` [13]. Since invariants protect the state of the program rather than the input and output of methods, they are treated a bit differently than pre- and postconditions in .NET Code Contracts. All calls to `Contract.Invariant` must reside in a special helper method that takes no parameters, returns **`void`** and is tagged with the `ContractInvariantMethodAttribute` type, which is an attribute provided by .NET Code Contracts. The helper method is specific to the C# class and is not defined in VDM-SL. Traditionally, it is named `ObjectInvariant`.

Invariants in .NET Code Contracts are only checked upon leaving any public method in contrast to VDM-SL where they must hold at all times. An invariant that has been satisfied so far can only be violated by manipulating the program state it is guarding. Therefore, in practice, it is only necessary to check invariants after program state mutations. By encapsulating all program state in C# properties, .NET Code Contracts will check the invariants after invoking a public property setter, as this is equivalent to calling a public setter method. Upon class instantiation, the class invariants are also checked after calling the public constructor.

Translating invariants

Let i be an invariant for type T, let e_{i} be the logical predicate of i, and let T_{inv} **:** T `->` **`bool`** be the self-contained function for i in VDM-SL. Then T becomes an appropriate type T' in C# and T_{inv} becomes a member of T' as the pure method T'_{inv}. The special `ObjectInvariant` helper method of T' calls `Contract.Invariant(`T'_{inv}`(this))`. T'_{inv} evaluates and returns e_{i}.

Type invariants Type invariants must be enforced in five places: parameters to functions and operations, result values from functions and operations, variable initialisers, assignments and value definitions [19].

Type invariants for method parameters are enforced as preconditions, that is, by calling `Contract.Requires` and checking that the type invariant predicate is satisfied. Type invariants for return values are enforced similarly as postconditions via `Contract.Ensures`.

All persistent state is stored in properties which are subject to invariant enforcement through the calls to `Contract.Invariant` carried out by the nested class

declarations. Temporary state in VDM-SL can be defined through the use of variables initialised in let-expressions and define-expressions. They are equivalent to local variable declarations in C#. Potential type invariants are enforced by calling the `Contract.Assert` method in .NET Code Contracts. It takes a predicate as input and evaluates it immediately.

VDM-SL values are translated to static read-only properties in C# that are initialised upon declaration. Therefore, type invariants on values only need to be enforced once. This is done by calling `Contract.Assert` from a static constructor in the class that contains the property. The property initialiser is always executed just before the static constructor [12]. So for a small VDM-SL extract that uses a type invariant such as:

```
types
    Pin = nat
    inv p == 0 <= p and p <= 9999;
```

the following C# code is produced:

```
public sealed class Pin : ICopyable<Pin>, IEquatable<Pin> {
    public int Value { get; }

    public Pin(int @value) { Value = @value; }

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(Value >= 0);
        Contract.Invariant(InvPin(Value));
    }

    [Pure]
    public static bool InvPin(int p) {
        Contract.Requires(p >= 0);
        return 0 <= p && p <= 9999;
    }
    // Equals, GetHashCode etc. have been omitted.
}
```

3.2 Collections

The .NET Standard Library provides the `ISet<T>`, `IList<T>` and `IDictionary<TDomain, TRange>` collection interfaces that correspond to sets, sequences and maps in VDM-SL [14]. Furthermore, the LINQ library defines extension methods to manipulate generic collections of type `IEnumerable<T>` (the base type of all collection types in .NET) which enable straightforward translation of most of VDM-SL's collection operations. For example, small VDM-SL expressions such as:

```
dunion m
iota i in set m & p
tl m
inverse m
```

are translated to the following LINQ queries in C#:

```
m.Cast<IEnumerable<T>>().Aggregate(Enumerable.Union).ToHashSet()
m.Single(i => p)
m.Skip(1).ToList()
m.ToDictionary(_ => _.Value, _ => _.Key)
```

Set comprehensions While C# does not offer a dedicated set comprehension construct like VDM-SL, it provides built-in language support for LINQ queries. A LINQ query is initiated with a **from**...**in** clause that specifies the collection to use as a starting point. The **from**...**in** clause is followed by an optional **where** clause, which filters the elements in the source collection, and a mandatory **select** clause, which projects the filtered elements into a new collection instance. Compound **from**...**in** clauses are useful for manipulating multiple collections in a single query. Hence, LINQ queries in C# emulate set comprehensions in VDM-SL.

Translating set comprehensions

Let C be a set comprehension with bindings $b_1, ..., b_n$, predicate expression p and projection expression e. Let the binding b_i be on the form s_i **in set** S_i in VDM-SL. Then C becomes a LINQ query C' in C# constituted by `(`**from** s_1 **in** S_1 `...` **from** s_n **in** S_n **where** p **select** e`).ToHashSet()`. If the set comprehension leaves out the predicate p, the **where** clause is omitted in the LINQ query.

A set comprehension that quantifies over multiple variables such as:

```
{mk_(x, y, z) | x, y, z in set {1, ..., 5} & x + y = z}
```

leads to the following rather compact LINQ query in C#:

```
(from x in Enumerable.Range(1, 5)
 from y in Enumerable.Range(1, 5)
 from z in Enumerable.Range(1, 5)
 where x + y == z
 select Tuple.Create(x, y, z)).ToHashSet()
```

Universal quantifications In addition to occurring in set comprehensions, set bindings also occur in quantified expressions such as universal quantifications, which check

whether a predicate holds for every element in a set. The translation of bindings in universal quantifications is similar to the one for set comprehensions: a **`from`**...**`in`** clause, compound if necessary. The **`select`** clause projects the elements of the bindings into Boolean values computed from the predicate. LINQ provides the `All` method for testing whether a predicate is satisfied for all elements in a single collection. In this case, it simply tests that all Boolean values are true.

Translating universal quantifications
Let Q be a universally quantified expression with bindings $b_1, ..., b_n$ and predicate expression p. Let the binding b_i be on the form s_i **`in set`** S_i in VDM-SL. Then Q becomes a LINQ query Q' in C# constituted by `(`**`from`** s_1 **`in`** S_1 **`...`** **`from`** s_n **`in`** S_n **`select`** p`).All(x => x)`, where `x => x` is a C# lambda expression for the Boolean identity function.

So for a small quantified expression in VDM-SL such as:

```
forall n in set s & n <= r
```

one gets the following LINQ query in C#:

```
(from n in s select n <= r).All(x => x)
```

3.3 Type aliases

A type alias that defines a type invariant is realised in C# by a wrapper class that contains a single property that holds the value of the underlying type subject to aliasing. By default, classes in C# exhibit referential equality and pass-by-reference semantics [12] so that it is just the object references, not the objects themselves, that are copied when they are passed around. This behaviour can be changed by overriding the `Equals` and `GetHashCode` methods to provide structural equality and implementing a `Copy` method to enable pass-by-value semantics like VDM-SL. The solutions proposed by Joshua Bloch [3] for overriding the `equals` and `hashCode` methods in corresponding Java classes have been adapted to C#.

Different wrapper classes of the same type are not interchangeable in C#, although this feature is achievable through type casting. In C#, both the explicit and implicit type cast operators can be overloaded to increase the number of valid type casts [12]. However, the class needs to overload the type cast operators for every other type alias that has been defined for the same underlying type in order to make them interchangeable. This leads to a combinatorial explosion of overloads, which is not desirable. It suffices to overload the type cast operators to and from the underlying type so that literals, for example integers, are automatically cast to instances of the class. By leaving the property public, it is always possible to retrieve the aliased value.

Translating type aliases

Let U be an alias for type T. Then T becomes an appropriate type T' in C# and U becomes a class U' that contains a single `Value` property of type T'. `Value` is initialised from a public constructor. U' overrides the `Equals` and `GetHashCode` methods to provide structural equality on the `Value` property. It also implements a `Copy` method that creates a deep copy of an instance of U'. Finally, it overloads the type cast operators from T' to U' and vice versa – the former by instantiating U' from an instance of T'; the latter by retrieving the wrapped instance of T' stored in the `Value` property in U'.

As an example of how to translate a VDM-SL type alias consider the example below:

```
types
    Pin = nat
    inv p == 0 <= p and p <= 9999;
```

For this example the following C# code is produced:

```
public sealed class Pin : ICopyable<Pin>, IEquatable<Pin> {
    public int Value { get; }

    public Pin(int @value) { Value = @value; }

    public static implicit operator Pin(int that)
        => new Pin(that);

    public static implicit operator int(Pin that)
        => that.Value;

    public Pin Copy() => new Pin(Value);

    public bool Equals(Pin that) {
        if (ReferenceEquals(null, that)) return false;
        if (ReferenceEquals(this, that)) return true;
        return Value == that.Value;
    }

    public override bool Equals(object that) {
        if (ReferenceEquals(this, that)) return true;
        return that is Pin && Equals((Pin) that);
    }

    public override int GetHashCode() {
        var result = 17;
        result = 31 * result + Value;
        return result;
    }
```

```
    // ObjectInvariant and InvPin have been omitted.
}
```

4 Comparison of .NET Code Contracts and OpenJML

This section introduces a VDM-SL specification that is used to analyse the performance of .NET Code Contracts and OpenJML. The VDM-SL specification models an algorithm that is used to obfuscate financial accounting district (FAD) codes – a six-digit number that identifies a retail branch. The FAD code example was originally used in [20] to analyse the performance of code generated traces executed using OpenJML. The obfuscation algorithm defined in the convert function, shown in Listing 1.1, takes a mapping of digits as input, and computes a permutation of all FAD codes so that no FAD code is mapped to itself.

```
values
    SIZE = 6;
    MAX = 10 ** SIZE - 1;
    DM1 : DigitMap =
        {1 |-> 9, 2 |-> 8, 3 |-> 7, 4 |-> 6, 5 |-> 0,
         6 |-> 4, 7 |-> 3, 8 |-> 2, 9 |-> 1, 0 |-> 5};
types
    DigitMap = inmap nat to nat
    inv m ==
        let digits = {0, ..., 9} in
            dom m = digits and rng m = digits
            and forall c in set dom m & m(c) <> c;

    FAD = nat
    inv f == f <= MAX
functions
    convert: FAD * DigitMap -> FAD
    convert(fad, dm) ==
        let digits = digitsOf(fad) in
            valOf([dm(digits(i)) | i in set inds digits])
    post RESULT <> fad;
```

Listing 1.1. The FAD model

The corresponding C# code is:

```
public static int Size { get; } = 6;

public static int Max { get; } = 10.IntPower(Size) - 1;

public static DigitMap Dm1 { get; } = new Dictionary<int, int>
{
    [1] = 9, [2] = 8, [3] = 7, [4] = 6, [5] = 0,
```

```
    [6] = 4, [7] = 3, [8] = 2, [9] = 1, [0] = 5
};

[Pure]
public static Fad Convert(Fad fad, DigitMap dm) {
    Contract.Requires(fad != null);
    Contract.Requires(dm != null);
    Contract.Ensures(Contract.Result<Fad>() != null);
    Contract.Ensures(
        PostConvert(fad, dm, Contract.Result<Fad>()));

    return Let(() => {
        var digits = DigitsOf(fad);
        return ValOf(
            (from i in Enumerable.Range(1, digits.Count)
                                 .ToHashSet()
             select dm.Value[digits[i - 1]]).ToList());
    });
}

[Pure]
public static bool PostConvert(Fad fad, DigitMap dm,
                               Fad result) {
    Contract.Requires(fad != null);
    Contract.Requires(dm != null);
    Contract.Requires(result != null);
    return !Equals(result, fad);
}

public sealed class DigitMap : ICopyable<DigitMap>,
                               IEquatable<DigitMap> {
    public DigitMap(Dictionary<int, int> @value) {
        Value = @value;
    }

    public Dictionary<int, int> Value { get; }

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(Value != null
            && Value.IsInjective()
            && Contract.ForAll(Value.Keys, _ => _ >= 0)
            && Contract.ForAll(Value.Values, _ => _ >= 0));
        Contract.Invariant(InvDigitMap(Value));
    }

    [Pure]
    public static bool InvDigitMap(Dictionary<int, int> m) {
        Contract.Requires(m != null && m.IsInjective()
            && Contract.ForAll(m.Keys, _ => _ >= 0)
```

```
            && Contract.ForAll(m.Values, _ => _ >= 0));

        return Let(() => {
            var digits = Enumerable.Range(0, 10).ToHashSet();
            return m.Keys.ToHashSet().SetEquals(digits)
                && m.Values.ToHashSet().SetEquals(digits)
                && (from c in m.Keys.ToHashSet()
                    select m[c] != c).All(_ => _);
        });
    }
    // Equals, GetHashCode etc. have been omitted.
}

public sealed class Fad : ICopyable<Fad>, IEquatable<Fad> {
    public Fad(int @value) { Value = @value; }

    public int Value { get; }

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(Value >= 0);
        Contract.Invariant(InvFad(Value));
    }

    [Pure]
    public static bool InvFad(int f) {
        Contract.Requires(f >= 0);
        return f <= Max;
    }
    // Equals, GetHashCode etc. have been omitted.
}
```

An exhaustive test checks that the `convert` function maps every FAD code to another FAD code. When the value of `SIZE` is 6, there are one million FAD codes to check.

This section carries out the exhaustive test in order to measure the computational performance of both the output from the C# and Java code generators. In C#, DbC elements are implemented and checked by .NET Code Contracts [17], whereas they are represented by JML annotations in Java, which can be checked using the OpenJML runtime-assertion checker [5].

The benchmark program runs the exhaustive test eight times and reports the average time spent per iteration. The first iteration is a warm-up iteration for the virtual machines – .NET CLR and Java HotSpot VM, respectively – and is not included in the result. The return value of `convert` is stored in a field variable whose final value is printed to the console in order to avoid the virtual machine performing dead code elimination[2]. Three experiments are carried out for each platform:

[2] See http://www.oracle.com/technetwork/java/hotspotfaq-138619.html.

Experiment I No contracts are checked. This corresponds to removing all DbC elements from the VDM-SL specification and executing the `convert` function. In C#, the definition of the `CONTRACTS_FULL` symbol is omitted to disregard all contracts. In Java, the program is compiled with the normal Java compiler in OpenJDK 7, which disregards all JML annotation comments. Thus, no contracts are emitted in the bytecode.

Experiment II Contracts are specified, but not checked during execution. This is the default mode of operation for production-ready software. In .NET Code Contracts, the contracts can be disabled via the Visual Studio extension. In Java, they can be disabled by omitting the '-ea' (enable assertions) command line argument. The program is compiled with the OpenJML compiler.

Experiment III Contracts are specified and checked during execution. This is the approach employed for software under development.

4.1 Results

Table 4.1 shows the results of the benchmarking experiments. The numbers denote the time spent executing a particular experiment[3]. They have been performed with `SIZE` values of 1 to 6, cf. Listing 1.1, yielding 10^{SIZE} iterations in an exhaustive test.

Size	.NET I [ms]	.NET II [ms]	.NET III [ms]	Java I [ms]	Java II [ms]	Java III [ms]
1	1	1	1	1	2	2
2	1	1	1	2	20	22
3	1	1	1	4	245	254
4	15	15	23	22	3,103	3,212
5	190	189	295	212	37,626	38,401
6	2,273	2,279	3,610	2,498	440,716	443,523

Table 1. Benchmark results.

Experiment I and II in .NET show that there is no notable difference between omitting the `CONTRACTS_FULL` symbol and configuring the Visual Studio extension to disable contracts at runtime. In fact, when contracts are disabled in Visual Studio, they are erased from the bytecode, just like omitting the `CONTRACTS_FULL` symbol would do. They complete the exhaustive test of a million iterations in about 2,300 milliseconds. In Experiment III, the contracts are emitted in the CIL bytecode and checked during

[3] Benchmarking has been performed on an HP desktop computer that runs a 64-bit edition of Windows 10 and is equipped with a 3.4 GHz Intel i7 Skylake processor and 16 GB RAM. The C# benchmarks have been run on .NET CLR 4.0, whereas the JML benchmarks have been run on Java HotSpot VM 1.8 after being built on OpenJDK 7 via an Ubuntu 14 virtual machine.

execution, which results in a performance overhead. It completes the exhaustive test in about 3,600 milliseconds, implying an overhead from code contracts of approximately 60%.

In Java, Experiment I performs about as well as .NET, completing the exhaustive test in about 2,500 milliseconds. However, Experiment II and III, which are compiled with OpenJML, suffer from a very large overhead. They complete the exhaustive test in about 440,700 and 443,500 milliseconds, respectively, making them approximately 175 times slower than Experiment I. This overhead has to be caused by the OpenJML compiler, otherwise the performance of Experiment II would be comparable to Experiment I, which it is clearly not. When assertions are disabled in Java, they should have no impact on the execution. This confirms the OpenJML performance results reported in [20].

As shown in Experiment III, when code contracts enabled, .NET Code Contracts is about 120 times faster than OpenJML. Fig. 4.1 shows the results in a logarithmically scaled graph.

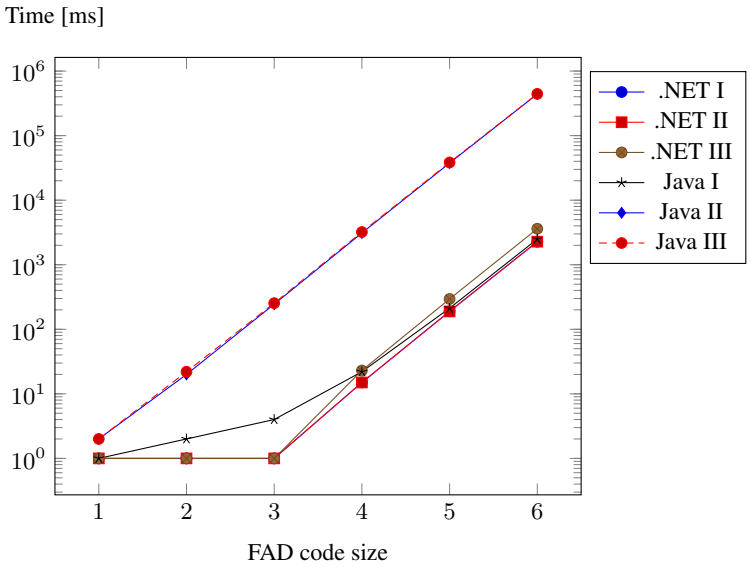

Fig. 2. Benchmark results visualised in a logarithmic data plot.

4.2 Validity of results

When inspecting the benchmark results, it should be taken into consideration that .NET CLR and Java HotSpot VM are two different environments and the Java and C# code generators use different approaches in some situations. For example, the C# code generator favours native language constructs and standard library methods over manually implemented utility methods like those shipped with the Java code generator runtime.

They also differ in how DbC elements are enforced: Type invariants such as bounds- and null-checking are implemented in C# as pre- and postconditions by .NET Code Contracts when they occur in method parameters and return values. The Java code generator, on the other hand, enforces the type invariants through JML assertions. Furthermore, type aliases are inlined by the Java code generator, but defined as separate wrapper classes by the C# code generator. Still, the difference in performance between .NET Code Contracts and OpenJML is so significant that it cannot be attributed to platform differences alone.

5 Conclusion and Future Work

In this paper, we have presented a handful of rules for translating core VDM-SL concepts to C# by utilising .NET Code Contracts and LINQ. They have been implemented in a transcompiler prototype for the Overture tool.

.NET Code Contracts performs significantly better than the OpenJML tool in the FAD code obfuscation model, which covers all kinds of DbC elements in VDM-SL. The benchmark results reveal that .NET Code Contracts is about 120 times faster than OpenJML, thus confirming the results of Jørgensen et al. [20] on the large overhead caused by OpenJML. Its impact is further demonstrated by the significant time gap between Java Experiment I and II.

The translation rules and transcompiler prototype targets only a subset of VDM-SL, leaving plenty of room for enhancements. For example, pattern matching and decomposition of tuples, records, sets etc. in VDM-SL are yet to be translated to C#. Basic object decomposition is to be introduced as a native feature of C# 7.0 [4].

Another extension is to support the object-oriented VDM++ dialect [10] and handle the translation of object aliasing, method overloading, multiple class inheritance and concurrency.

Acknowledgements We would like to thank the anonymous referees for valuable input on this work. The authors would also like to thank Aslan Askarov and Erik Ernst for input on this work.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
2. Barnett, M., Leino, R.M., Schulte, W.: The Spec# Programming System: An Overview. In: CASSIS Proceedings (October 2004)
3. Bloch, J.: Effective Java. Addison-Wesley, second edn. (2008)
4. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML Tools and Applications. Intl. Journal of Software Tools for Technology Transfer 7, 212–232 (2005)

[4] See https://blogs.msdn.microsoft.com/dotnet/2016/08/24/whats-new-in-csharp-7-0/.

5. Cok, D.R.: OpenJML: JML for Java 7 by Extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-20398-5_35
6. Dalvandi, M., Butler, M., Rezazadeh, A.: From Event-B Models to Dafny Code Contracts, pp. 308–315. Springer International Publishing, Cham (2015)
7. Diswal, S.P.: Transcompilation of VDM-SL to C#. Master's thesis, Aarhus University, Department of Computer Science (June 2016)
8. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
9. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
10. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (April 2013)
11. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370. Springer (2010)
12. Microsoft: C# Language Specification 5.0 (2013)
13. Microsoft: Code Contracts User Manual. https://www.microsoft.com/en-us/research/project/code-contracts/ (2013), accessed Sep 1st 2016
14. Microsoft: Language-Integrated Query (LINQ) (C#). http://msdn.microsoft.com/da-dk/library/mt693024.aspx (2015)
15. Petzold, C.: .NET Book Zero (2007)
16. Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code generation for Event-B. International Journal on Software Tools for Technology Transfer pp. 1–22 (2015)
17. Skeet, J.: C# in Depth, Second Edition. Manning Publications (2010)
18. Strauss, D.: C# Code Contracts Succinctly. Syncfusion Inc. (2016)
19. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML annotated Java (January 2016 Submitted to the International Journal on Software Tools for Technology Transfer (STTT))
20. Tran-Jørgensen, P.W., Larsen, P.G., Battle, N.: Using JML-based Code Generation to Enhance the Test Automation for VDM Models. In: The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 79–93. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
21. Visser, J., Oliveira, J.N.F., Barbosa, L., Ferreira, J.F., Mendes, A.: CAMILA revival: VDM meets Haskell. In: 1st Overture Workshop. University of Newcastle TR series (2005)