Uniwersytet Wrocławski
Instytut Informatyki
Informatyka

Praca magisterska

Dynamiczna analiza poprawności programów na platformę .NET z wykorzystaniem bibliotek do profilowania

Adam Szeliga

Promotor: dr Wiktor Zychla

Wrocław, 2011

Spis treści

Rozdział 1

Wstęp

W niniejszej pracy opisano budowę oraz zasadę działania biblioteki AsProfiled umożliwiającej kontrolę poprawności działania dowolnego programu działającego w obrębie platformy .NET.
Poprawność ta badana jest poprzez weryfikację kontraktów nałożonych na poszczególne części programów, w tym wypadku, funkcji (metod). Ten rodzaj weryfikacji nazywany jest programowaniem kontraktowym. Pojęcie to zostało wprowadzone przez Bertranda Meyera w odniesieniu do języka programowania Eiffel.

Przedstawiona biblioteka została napisana w języku C++ przy wykorzystaniu mechanizmów związanych z technologią COM. Należy wspomnieć, iż w kontekście programowania kontraktowego jest to rozwiązanie, jak do tej pory, unikalne.

1.1 Platforma .NET i CLR

Zasada działania opisywanego rozwiązania mocno opiera się na mechanizmach służących do profilowania aplikacji. Teoria profilowania została przedstawiona w kolejnych rozdziałach. Jak każde podobne rozwiązanie tak i to jest silnie związane ze środowiskiem uruchomieniowym. W tym przypadku jest to platforma .NET stworzona przez firmę Microsoft i przeznaczona dla systemów z rodziny Windows.
Technologia ta nie jest związana z żadnym konkretnym językiem programowania, a programy mogą być pisane w jednym z wielu języków – na przykład C++/CLI, C#, J#, Delphi 8 dla .NET, Visual Basic .NET. Zadaniem kompilatorów jest translacja programów wyrażonych w w/w języków na język pośredni CIL (wcześniej MSIL). Dopiero tak przygotowane programy mogą być wykonane na maszynie wirtualnej CLR, która to jest środowiskiem uruchomieniowym platformy .NET. Taka konstrukcja pozwoliła rozszerzyć zakres działania utworzonego rozwiązania na wszystkie języki

programowania w obrębie tej platformy, pod warunkiem, że dany język wspiera konstrukcje programowe zwane atrybutami.

1.2 Założenia

W celu zapewnienia jak największej użyteczności, zostały przyjęte minimalne założenia co do funkcjonalności jakie muszą być zawarte w bibliotece. Wszystkie z nich zostały szczegółowo opisane w rozdziale czwartym, jednak wprowadzamy je już teraz, aby w dalszej uzasadnić decyzje podjęte przy konstrukcji kolejnych etapów aplikacji.

- biblioteka musi śledzić proces wykonywania programu po jego uruchomieniu
- w celu weryfikacji poprawności programu musi być możliwość zdefiniowana kontraktu
- musi być możliwość odczytania zadanego kontraktu
- aplikacja musi wiedzieć, dla której metody ma się odbyć weryfikacja
- aplikacja musi umieć odczytać argumenty przekazywane do badanych metod
- aplikacja musi zachowywać stan początkowy argumentów metody do momentu jej zakończenia
- aplikacja musi być w stanie odczytać wartości zwracane z badanych metod

W kolejnych rozdziałach opisane w jaki sposób każde z powyższych założeń zostało spełnione. Nie przewidziano żadnych założeń co do wymagań pozafunkcjonalnych, co oznacza, iż takie parametry jak szybkość działania aplikacji czy bezpieczeństwo rozwiązania, nie były przedmiotem zainteresowania.

Rozdział 2

Programowanie kontraktowe

W tym rozdziale została przybliżona specyfika programowania kontraktowego. Programowanie kontraktowe jest metodologią sprawdzania poprawności oprogramowania.

2.1 Historia

Koncepcja ta ma korzenie w pracach nad formalną weryfikacją programów, formalną specyfikacją oraz związanych z logiką Hoara. Wszystkie z powyższych dążą do dowodzenia poprawności programów komputerowych, a tym samym przyczyniają się podnoszeniu ich jakości. Nie inaczej jest w przypadku programowanie kontraktowego. Po raz w obecnej postaci wprowadził je Bertrand Meyer w 1986 roku przy okazji wprowadzenia na rynek projektu języka programowania Eiffel. Do dnia dzisiejszego powstało wiele różnych różnych implementacji tej koncepcji. Cześć języków programowania ma wbudowane mechanizmy pozwalające na definiowanie i sprawdzanie poprawności kontraktów. Do tej grupy zaliczamy:

- Cobra
- Eiffel
- D
- języków opartych na platformie .NET w wersji 4.0

Drugą grupę stanowią języki dla których powstały nakładki umożliwiające ten rodzaj weryfikacji. Ta grupa jest znacznie obszerna i obejmuje większość znaczących języków programowania, takich jak :

- C/C++,

- C#
- Java
- Javascript
- Perl
- Python
- Ruby

Omawiana biblioteka należy do drugiej grupy rozwiązań.

2.2 Opis

Ten rodzaj programowania zakłada, że elementy programu powinny odnosić się do siebie na zasadzie kontraktów, czyli:

- Każdy element powinien zapewniać określoną funkcjonalność i wymagać ściśle określonych środków do wykonania polecenia.
- Klient może użyć funkcjonalności, o ile spełni zdefiniowane wymagania.
- Kontrakt opisuje wymagania stawiane obu stronom.
- Element zapewniający funkcjonalność powinien przewidzieć sytuacje wyjątkowe, a klient powinien je rozpatrzyć.

Koncepcja ta polega na zawieraniu swego rodzaju umowy pomiędzy dostawcą funkcjonalności i klientami. W ogólnym przypadku poprzez dostawców rozumiemy klasy lub metody zawarte w programie, klientem zaś jest każdy kto z tych encji korzysta.
Dla danej klasy kontrakt definiowany jest jako inwariant, to znaczy, warunek jaki musi być spełniony przed i po wywołaniu dowolnej publicznej metody w obrębie tej klasy. Z kolei dla metod kontrakt definiowany przy pomocy warunków początkowego i końcowego, gdzie ten pierwszy specyfikuje jakie założenia powinny być spełnione w momencie wywołania metody, a drugi określa stan aplikacji po jej zakończeniu.

Rozwiązanie, które jest tu opisywane skupia się na drugim rodzaju kontraktów. W języku programowania Eiffel, skąd wywodzi się cała idea, kontrakty opisywane są w sposób następujący:

Listing 2.1: Programowanie kontraktowe

```
NazwaMetody (deklaracja argumentów) is
require
 -- warunek początkowy
do
 -- ciało metody
ensure
 -- warunek początkowy
end
```

Przy budowie prezentowanej aplikacji wykorzystano cechę szczególną platformy .NET, a w szczególności języka C#, jaką jest możliwość dekorowania metod atrybutami. Ten element języka będzie dokładniej opisany w dalszej części pracy, przy okazji przedstawiania szczegółów implementacyjnych. W tym momencie wystarczy przyjąć, iż atrybuty te stają się częścią meta informacji o danej metodzie, co z kolei może być wykorzystywane przy jej inspekcji. Właśnie ta cecha została wykorzystana w rozważanej aplikacji. Dla ilustracji, poniżej została zademonstrowana ogólna postać zapisu kontraktów w języku C#:

Listing 2.2: Definicja kontraktu

```
[NazwaAtrybutuDefiniującegoKontrakt( warunek początkowy, warunek
    końcowy )]
NazwaMetody( deklaracja argumentów )
{
  -- definicja metody
}
```

Rozdział 3

Profilowanie

3.1 Profilowanie – opis

Profiling, in this document, means monitoring the performance and memory usage of a program, which is executing on the Common Language Runtime (CLR). This document details the interfaces, provided by the Runtime, to access such information. Typically, a very limited audience will use these APIs – developers of profiling tools. Just to give the flavor, a typical use for profiling is to measure how much time (elapsed, or wall-clock, and/or CPU time) is spent within each routine, or within all code that is executed from a given root routine. To do this, a profiler asks the Runtime to inform it whenever execution enters or leaves each routine; the profiler notes the wall-clock and CPU time for each such event, and accumulates the results at the end of the program. Note that the term routine is being in this document to indicate a section of code that has an entry point and an exit point. Different languages use different names for this same concept – function, procedure, method, co-routine, subroutine, etc. Profiling a CLR program requires more support than profiling conventionally compiled machine code. This is because the CLR has introduced new concepts such as application domains, garbage collection, managed exception handling, JIT compilation of code (converting Microsoft Intermediate Language into native machine code) etc that the existing conventional profiling mechanisms are unable to identify and provide useful information. The profiling APIs provide this missing information in an efficient way that causes minimal impact on the performance of the CLR. Note that JIT-compiling routines at runtime provide good opportunities, as the APIs allow a profiler to change the in-memory MSIL code stream for a routine, and then request that it be JIT-compiled anew. In this way, the profiler can dynamically add instrumentation code to particular routines that need deeper investigation. Although this approach is possible in conventional scenarios, it's much easier to do this for the CLR.

Expose information that existing profilers will require for a user to determine and analyze performance of a program run on the CLR. Specifically: .. Common Language Runtime startup and shutdown events .. Application domain creation and shutdown events .. Assembly loading and unloading events .. Module load/unload events .. Com VTable creation and destruction events .. JIT-compiles, and code pitching events .. Class load/unload events .. Thread birth/death/synchronization .. Routine entry/exit events .. Exceptions .. Transitions between managed and unmanaged execution .. Transitions between different Runtime contexts .. Information about Runtime suspensions Profiling Page 9 .. Information about the Runtime memory heap and garbage collection activity • Callable from any COM-compatible language • Efficient, in terms of CPU and memory consumption – the act of profiling should not cause such a big change upon the program being profiled that the results are misleading • Useful to both sampling and non-sampling profilers. [A sampling profiler inspects the profilee at regular clock ticks – maybe 5 milliseconds apart, say. A nonsampling profiler is informed of events, synchronously with the thread that causes them]

3.2 Typy profilowania

3.2.1 Instrumentacja

3.2.2 Próbkowanie

3.3 Profilowanie w środowisku .NET

3.4 Profilowanie a dynamiczna weryfikacja programów

... jakiś tekst ...

Rozdział 4

Omówienie funkcjonalności biblioteki

W tym rozdziale szczegółowo opisane zostały funkcjonalności jakie udostępnia biblioteka.

4.1 Inspekcja nadzorowanego programu

Jak to zostało wspomniane we wcześniejszych rozdziałach, aplikacja weryfikująca kontrakty ma postać biblioteki COM i jako taka musi być wcześniej zarejestrowana w systemie. Do tego celu używana jest aplikacja o nazwie regsrv32.exe, która to jest częścią narzędzi dostarczanych wraz z platformą .NET. Zadaniem tego narzędzia jest pobranie identyfikatora biblioteki i umieszczenie w rejestrze systemu klucza przechowującego ten identyfikator oraz ścieżkę w systemie pliku pod która znajduje się biblioteka.

Rozpoczęcie procesu profilowania/weryfikacji aplikacji odbywa się poprzez uruchomienie programu z linii poleceń w odpowiednio przygotowanym środowisku. Etap ten polega na ustawieniu zmiennych środowiskowych, instruujących maszynę wirtualną CLR, aby ta wysyłała powiadomienia na temat zdarzeń zachodzących wewnątrz uruchamianej aplikacji. Proces ten wygląda w sposób następujący:

```
SET COR_ENABLE_PROFILING=1
SET COR_PROFILER={GUID}
```

Powyższe zmienne są następnie odczytywane przez środowisko uruchomieniowe. Pierwsza z nich informuje maszynę wirtualną, że ta powinna przesyłać informacje o zdarzeniach do biblioteki, której położenie określanie jest przy wykorzystaniu identy-

fikatora GUID.

Liczba i rodzaj wysyłanych powiadomień określany jest wewnątrz biblioteki profilującej. W szczegółach temat ten opisany jest w kolejnym rozdziale.

4.2 Kontrakty

Podstawowym elementem, dzięki któremu możliwa jest weryfikacja metod, jest oczywiście możliwość definiowania kontraktu. Jak już zostało wspomniane we wcześniejszych rozdziałach kontrakty definiujemy za pomocą atrybutów.

Atrybuty są to znaczniki o charakterze deklaracyjnym zawierające informację o elementach programu (np. klasach, typach wyliczeniowych, metodach) przeznaczoną dla środowiska wykonania programu. Co jest w tym kontekście istotne to iż są one pamiętane jako meta-dane elementu programu.

Definicja atrybutów jest jedynym elementem, wchodzącym bezpośrednio w skład omawianego rozwiązania, który musi znajdować się po stronie weryfikowanej aplikacji.

Jak już zostało wspomniane atrybuty określające kontrakt mają postać:
AsContract(Warunek początkowy, Warunek końcowy)

Oba warunki zdefiniowane są poprzez pewne, określone przez użytkownika wyrażenie. Te z kolei mają postać określoną przez zadaną gramatykę, której definicję przedstawiono w następnym rozdziale dotyczących implementacji. Należy tu jednak wspomnieć, iż oba warunki zapisywane są jako łańcuchy znakowe, tak więc konieczne jest ich parsowanie, w celu otrzymania drzewa rozbioru takiego wyrażenia. Kolejnym krokiem jest ewaluacja tego drzewa, aby można było określić czy udekorowana metoda spełnia nałożony na nią kontrakt.

4.3 Metadane

Metadane w kontekście platformy .NET, to dodatkowe informacje opisujące składowe języka. Są usystematyzowanym sposobem reprezentowania wszystkich informacji, których CLI używa do lokalizowania i ładowania klas, ułożenia obiektów w pamięci, wywoływania metod, translacji języka CIL do kodu natywnego.

Dane te, emitowane przez kompilator, przechowywane są wewnątrz każdego wykonywalnego programu w postaci binarnej.

4.3.1 Reprezentacja metadanych

W ramach systemu Windows zdefiniowany jest format plików wykonywalnych - PE (eng. Portable Executables), określający strukturę jaką musi posiadać każdy program, aby mógł być uruchomiony w systemie. Aplikacje przeznaczone na platformę .NET naturalnie również muszą być zorganizowane w sposób zgodny z tym standardem.

Jednym z pól w ramach nagłówka CLI jest offset (RVA) określający położenie zbioru metadanych w ramach pliku wykonywalnego czy biblioteki.

Metadane przechowują informacje na temat typów definiowanych w ramach programu (klasy, struktury, interfejsy), globalnych funkcji i zmienny. Każda z tych abstrakcyjnych encji niesie ze sobą wartość wartości typu mdToken (metadata token). Jest ona używana przez mechanizmy odczytujące metadane do identyfikacji informacji na temat encji w ramach określonego zasięgu.

Token metadanych ma postać czterobajtowej wartości. Najbardziej znaczący bajt określa typ tokenu, pozostałe określają położenie pozostałych informacji w tablicy metadanych. Dla przykładu, wartość 1 przechowywana w MSB (most significant byte) oznacza, iż token jest typu mdTypeRef, który oznacza referencję do typu, a informacje na jego temat są przechowywane w tablicy TypeRef.

Pozostałe, mniej znaczące bajty, oznaczają identyfikator rekordu (record identifier - RID) i zawierają w sobie indeks do wiersza w/w tablicy, która określona jest przez wartość najbardziej znaczącego bajtu. Przykładowo, token o wartości 0x02000007 odnosi się do siódmego wiersza tablicy TypeRef. Podobnie, wartość 0x0400001A oznacza odwołanie do wiersza dwudziestego szóstego tabliy FieldDef. Wiersz zerowy każdej z powyższych tablic nigdy nie zawiera w sobie danych, więc jeśli identyfikator RID jest równy zeru, to znaczy to, że token jest pusty, ma wartość nil. Taki token zdefiniowany jest dla każdego typu encji, np. wartość 0x10000000 określa pusty token mdTypeRefNil.

W poniższej tabeli znajdują się wszystkie typy tokenów, typy które opisują oraz nazwy tablic metadanych. Wszystkie tokeny są pochodnymi typu bazowego - mdToken.

Token Type	Metadata Table	Abstraction
mdModule	Module	Module: A compilation unit, an executable, or some other development unit, deployment unit, or run-time unit. It is possible (though not required) to declare attributes on the module as a whole, including a name, a GUID, custom attributes, and so forth.
mdModuleRef	ModuleRef	Module reference: A compile-time reference to a module, which records the source for type and member imports.
mdTypeDef	TypeDef	Type declaration: Declaration of either a runtime reference type (class or interface) or a value type.

mdTypeRef	TypeRef	Type reference: Reference to either a runtime reference type or a value type. In a sense, the collection of type references in a module is the collection of compile-time import dependencies.
mdMethodDef	MethodDef	Method definition: Definition of a method as a member of a class or interface, or as a global module-level method.
mdParamDef	ParamDef	Parameter declaration: Definition of an optional data structure that stores additional metadata for the parameter. It is not necessary to emit a data structure for each parameter in a method. However, when there is additional metadata to persist for the parameter, such as marshaling or type-mapping information, an optional parameter data structure can be created.
mdFieldDef	FieldDef	Field declaration: Declaration of a variable as a data member of a class or interface, or declaration of a global, module-level variable.
mdProperty	Property	Property declaration: Declaration of a property as a member of a class or interface.
mdEvent	Event	Event declaration: Declaration of a named event as a member of a class or interface.
mdMemberRef	MemberRef	Member reference: Reference to a method or field. A member reference is generated in metadata for every method invocation or field access that is made by any implementation in the current module, and a token is persisted in the Microsoft intermediate language (MSIL) stream. There is no runtime support for property or event references.

mdIfaceImpl	IfaceImpl	Interface implementation: A specific class's implementation of a specific interface. This metadata abstraction enables the storing of information that is the intersection of that which is specific to neither the class nor the interface.
mdMethodImpl	MethodImpl	Method implementation: A specific class's implementation of a method that is inherited using interface inheritance. This metadata abstraction enables information to be persisted that is specific to the implementation rather than to the contract. Method declaration information cannot be modified by the implementing class.
mdCustomAttribute	CustomAttribute	Custom attribute: An arbitrary data structure associated with any metadata object that can be referenced with an mdToken. (An exception is that custom attributes themselves cannot have custom attributes.)
mdPermission	Permission	Permission set: A declarative security permission set associated with mdTypeDef, mdMethodDef, and mdAssembly. For more information, see Adding Declarative Security Support.
mdTypeSpec	TypeSpec	Type constructor: A method that obtains a token for a type (such as a boxed value type) that can be used as input to any MSIL instruction that takes a type.
mdSignature	Signature	Stand-alone signature: A local variable signature in the portable executable (PE) file or a method signature that is passed to an MSIL instruction.
mdString	String	User string: A string that is passed to an MSIL instruction.

4.3.2 Interpretacja metadanych

Do zaimplementowania procesu weryfikacji aplikacji przy obranym tutaj podejściu zachodzi potrzeba interpretacji metadanych. W tym celu musimy wiedzieć jaką postać mają poszczególne typy metadanych i w jaki sposób należy je interpretować. Proces ten rozpoczyna się od pobrania wartości tokenu odpowiedniego typu. W ogólnym przypadku należałoby rozważyć wszystkie ich rodzaje, w tej sytuacji możemy się jednak ograniczyć do tych, które zostały wykorzystane w bibliotece :

Tokeny odpowiedniego rodzaju uzyskiwane są poprzez wywołania odpowiednich metod na obiektach implementujących określone interfejsów. Dzięki informacją zawartych w metadanych możliwy jest odczyt niezbędnych informacji. W opisywanej bibliotece zaimplementowano uzyskiwanie następujących danych:

Metody: GetMethodName GetCallingConvention GetArgumentsCount GetReturnValue ReadArgumentsValues GetAttribute

Parametry: GetType

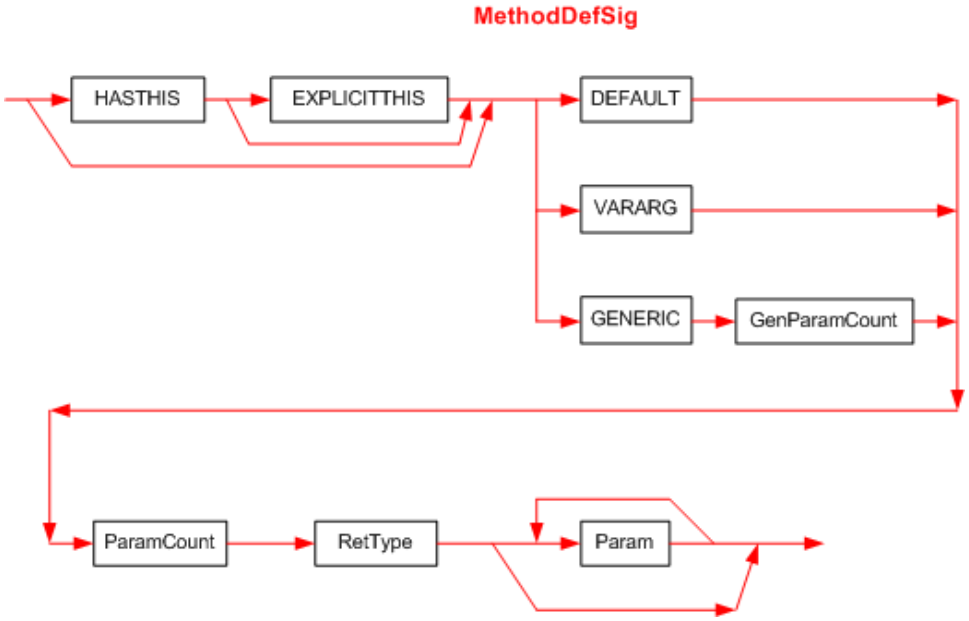
Typy: GetName

Atrybuty: ParseAttributeMetaData

4.4 Odczytywanie wartości obiektów

Metadane odgrywają kluczową rolę w procesie odczytywania wartości argumentów funkcji w czasie jej wykonywania. Dzięki tym informacjom można wyznaczyć typ dowolnego obiektu, w szczególności argumentów funkcji. Dzięki temu możliwe jest odczytanie aktualnej wartości argumentu w kontekście działającego programu. Typ definiuje sposób ułożenia obiektu w pamięci oraz określa jego strukturę wewnętrzną. Aktualny stan obiektu jest uzyskiwany poprzez inspekcje danych binarnych, do których odniesienie znajduje się wewnątrz metadanych obiektu.

The word signature is conventionally used to describe the type info for a function or method; that is, the type of each of its parameters, and the type of its return value. Within metadata, the word signature is also used to describe the type info for fields, properties, and local variables. Each Signature is stored as a (counted) byte array in the Blob heap. There are several kinds of Signature, as follows: • MethodRefSig (differs from a MethodDefSig only for VARARG calls) • MethodDefSig • FieldSig • PropertySig • LocalVarSig • TypeSpec • MethodSpec The value of the first byte of a Signature 'blob' indicates what kind of Signature it is. Its lowest 4 bits hold one of the following: C, DEFAULT, FASTCALL, STDCALL, THISCALL, or VARARG (whose values are defined in §23.2.3), which qualify method signatures; FIELD, which denotes a field signature (whose value is defined in §23.2.4); or PROPERTY, which denotes a property signature (whose value is defined in §23.2.5). This subclause defines the binary 'blob' format for each kind of Signature. In the syntax diagrams that accompany many of the definitions, Partition II 153 shading is used to combine into



Rysunek 4.1:

a single diagram what would otherwise be multiple diagrams; the accompanying text describes the use of shading. Signatures are compressed before being stored into the Blob heap (described below) by compressing the integers embedded in the signature. The maximum encodable integer is 29 bits long, 0x1FFFFFFF. The compression algorithm used is as follows (bit 0 is the least significant bit): • If the value lies between 0 (0x00) and 127 (0x7F), inclusive, encode as a one-byte integer (bit 7 is clear, value held in bits 6 through 0) • If the value lies between 28 (0x80) and 214 – 1 (0x3FFF), inclusive, encode as a 2-byte integer with bit 15 set, bit 14 clear (value held in bits 13 through 0) • Otherwise, encode as a 4-byte integer, with bit 31 set, bit 30 set, bit 29 clear (value held in bits 28 through 0) • A null string should be represented with the reserved single byte 0xFF, and no following data

Struktura elementów określona jest w ECMA Partition 2

This diagram uses the following abbreviations: HASTHIS = 0x20, used to encode the keyword instance in the calling convention, see §15.3 EXPLICITTHIS = 0x40, used to encode the keyword explicit in the calling convention, see §15.3 DEFAULT = 0x0, used to encode the keyword default in the calling convention, see §15.3 VARARG = 0x5, used to encode the keyword vararg in the calling convention, see §15.3 GENERIC = 0x10, used to indicate that the method has one or more generic parameters. The first byte of the Signature holds bits for HASTHIS, EXPLICITTHIS and calling convention (DEFAULT, VARARG, or GENERIC). These are ORed together. GenParamCount is the number of generic parameters for the method. This is a compressed int32. [Note: For generic methods, both MethodDef and MemberRef shall include the GENERIC calling convention, together with GenParamCount; these are

significant for binding—they enable the CLI to overload on generic methods by the number of generic parameters they include. end note] ParamCount is an integer that holds the number of parameters (0 or more). It can be any number between 0 and 0x1FFFFFFF. The compiler compresses it too (see Partition II Metadata Validation) – before storing into the 'blob' (ParamCount counts just the method parameters – it does not include the method's return type) The RetType item describes the type of the method's return value (§23.2.11) The Param item describes the type of each of the method's parameters. There shall be ParamCount instances of the Param item (§23.2.10)

4.5 Wartości początkowe

Termin wartości początkowe odnosi się do stanu argumentów w momencie wywołania metody. Zgodnie z założeniami z rozdziału pierwszego, biblioteka powinna udostępniać funkcjonalność definiowania kontraktów złożonych z odwołań do początkowych wartości obiektów. Naturalnie, stan obiektów może ulec zmianie w czasie działania metody, należało więc przedsięwziąć kroki umożliwiające tego rodzaju odwołania do początkowych wartości obiektów.

Jedyną sytuacją jaką należało rozważyć, jest w przypadku kontraktów, w których odwołania do wartości początkowych ma miejsce w warunkach końcowych. Rzecz jasna, odwołania do tych wartości nie mają sensu w przypadku warunków początkowych, gdyż te ewaluowane są zanim sterowanie dojdzie do momentu wykonywania instrukcji wewnątrz metody, a które mogłyby zmodyfikować stan obiektu. Z drugiej strony, co wynika ze specyfiki otrzymywanych powiadomień, odczytywanie argumentów jest możliwe tylko w momencie wywoływania metody. Co za tym idzie, niezbędne jest zachowanie stanu obiektu w tym momencie i odwoływanie się do niego w czasie weryfikacji warunków początkowych. Dodatkowo, nie wystarczy zachowanie referencji do obiektu. Wynika to z faktu, iż pomimo przekazywanie argumentów do metody odbywa się poprzez kopiowanie, to kopiowana jest tylko referencja do obiektu, a nie sam obiekt. Oczywiście kopia referencji dalej wskazuje na ten sam obiekt, więc zmieniając jego stan przy jej użycia, zmieniany jest oryginalny obiekt. Z tego powodu niezbędne okazuje się kopiowanie poszczególnych wartości do których występuje odwołanie w warunkach końcowych. Konieczne jest więc przeprowadzanie preprocessingu, tzn. w chwili otrzymania notyfikacji o wywołaniu metody przeprowadzona zostaje analiza zarówno warunków początkowych (co jest jasne) i warunków końcowych kontraktu. W czasie tej analizy warunek końcowy sprawdzany jest pod kątem występowania elementów odnoszących się do stanu początkowego argumentów funkcji. Następnie odpowiednie argumenty poddawane są inspekcji, a następnie wyłuskiwana jest wartość składowej obiektu, do którego odniesienie znajduje się w warunku końcowym. Ta wartość zapisywana jest w pamięci podręcznej aplikacji, do której dostęp jest możliwy do momentu otrzymania powiadomienia o wyjściu z zasięgu weryfikowanej metody,

kiedy to może zostać użyta do weryfikacji warunku końcowego.

4.6 Wartości zwracane

Kolejnym elementem jest możliwość weryfikacji kontraktów zawierających w sobie odniesienia do wartości zwracanej przez metodę. Takie odniesienia mają tylko sens przy warunkach końcowych, po odebraniu notyfikacji o zdarzeniu opuszczenia metody.

4.7 Ograniczenia

Rozdział 5

Szczegóły implementacji

W tym rozdziale opisano szczegóły dotyczące implementacyjne biblioteki.

5.1 Atrybuty jako kontrakty

Kontrakty definiowane są jako atrybuty, którymi dekorowane są metody. Definicja atrybutów ogranicza się do elementarnej klasy, której cała definicja zawarta jest w następującym bloku kodu:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public class AsContractAttribute : Attribute
{
  public AsContractAttribute(string preCondition, string
      postCondition)
  { }

  public string PostCondition { get; set; }

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

Linia nr 1 określa, iż atrybut może być przypisywany tylko do metod i może występować tylko jeden raz.

Centralnym elementem tej klasy jest dwuparametrowy konstruktor, przyjmujący dwa napisy jako parametry. Te właśnie napisy określają kontrakt.

Wyrażenia opisujące kontrakty muszę być obliczalne do jednej z dwóch wartości: prawdy albo fałszu, co oznacza odpowiednio, że kontrakt został lub też nie został spełniony.

5.2 Gramatyka kontraktów

Możliwy zbiór wyrażeń wyrażalnych poprzez kontrakty definiowany jest poprzez gramatykę bezkontekstową. Gramatyka ta została wykorzystana w aplikacji GOLD Parsing System, która na tej podstawie generuje tablicę stanów dla deterministycznego automatu skończonego. Poniżej znajduje się jej definicja :

Listing 5.1: Gramatyka kontraktów

```
"Start Symbol" = <Program>

! Sets

{ID Head}      = {Letter} + [_]
{ID Tail}      = {Alphanumeric} + [_]
{String Chars} = {Printable} + {HT} - [\\]
{Number Without Zero} = {Number} - [0]

! Terminals

Identifier = {ID Head}{ID Tail}*(.{ID Head}{ID Tail}*)*
StringLiteral = '"' ( {String Chars} | '\' {Printable} )* '"'
DecimalNumber = {Number Without Zero}{Number}* | {Number}
BooleanLiteral = 'true' | 'false'
ReturnValue = '@returnValue'(.{ID Head}{ID Tail}*)*
InitialValue = '^'{ID Head}{ID Tail}*(.{ID Head}{ID Tail}*)*

! Rules

<Program>           ::= <Boolean Exp>
<Boolean Exp>       ::= <Boolean Exp> <Boolean Operator> <Cmp
    Exp>
                    |   <Cmp Exp>

<Boolean Operator>  ::= '||'
                    |   '&&'

<Cmp Exp>           ::= <Cmp Exp> <Cmp Operator> <Add Exp>
                    |   <Add Exp>
<Cmp Operator>      ::= '>'
                    |   '<'
                    |   '<='
                    |   '>='
                    |   '=='
                    |   '!='

<Add Exp>           ::= <Add Exp> <Add Operator> <Mult Exp>
                    |   <Mult Exp>
<Add Operator>      ::= '+'
                    |   '-'
```

```
<Mult Exp>          ::= <Mult Exp> <Mult Operator> <Negate Exp>
                    |   <Bit Exp>
<Mult Operator>     ::= '*'
                    |   '/'
<Bit Exp>           ::= <Bit Exp> <Bit Operator> <Negate Exp>
                    |  <Negate Exp>
<Bit Operator>      ::= '&'
                    |   '|'

<Negate Exp>        ::= <Negate Operator> <Value>
                    |  <Value>
<Negate Operator>   ::= '-'
<Value>             ::= Identifier
                    |  StringLiteral
                    |  DecimalNumber
                    |  '(' <Boolean Exp> ')'
                    |  BooleanLiteral
                    |  ReturnValue
                    |  InitialValue
```

Poniżej przedstawiono kilka przykładów wyrażeń, które mogą być zbudowane przy użyciu reguł zawartych w gramatyce:

5.3 Omówienie i implementacja interfejsów

AsProfiled jak każda bibliotek typu COM udostępnia swoją funkcjonalność poprzez interfejsy. Niezależnie od przeznaczenia biblioteki musi ona przynajmniej implementować interfejs IUnknown, dzięki któremu możliwe jest uzyskanie uchwytu do pozostałych interfejsów definiujących określone funkcjonalności. W tym przypadku konieczne jest uzyskanie uchwytu do obiektu implementującego interfejs ICorProfilerCallback2. To poprzez niego odbywa cała komunikacja pomiędzy maszyną CLR a biblioteką AsProfiled. Interfejs ten zawiera kilkadziesiąt metod, poprzez które maszyna wirtualna może powiadomić odbiorcę o zdarzeniach zachodzących w obrębie profilowanego programu. Pełna jego definicja znajduje się w załączniku (A). Na potrzeby tej pracy wystarczające jest omówienie dwóch z nich, mianowicie :

```
STDMETHOD(Initialize)(IUnknown *pICorProfilerInfoUnk);
STDMETHOD(Shutdown)();
```

Naturalnie, implementując dowolny interfejs, niezbędne jest zdefiniowanie każdej zawartej w nim metody, jednak w przypadku metod, które nie stanowią przedmiotu zainteresowania wystarczające jest zwrócenie rezultatu świadczącego o poprawnym wykonaniu metody. W tym przypadku, taką wartością jest S_OK (0), standardowo określającą poprawne zakończenie wykonywania funkcji.

Metody listingu (1), jak sama nazwa wskazuje, są wywoływane podczas inicjalizacji biblioteki i w momencie zakończenia wykonywania programu. W ramach funkcji Shutdown() zwyczajowo zwalniane są uchwyty do obiektów wykorzystywanych w bibliotece. Przeciwnie do niej, w metodzie Initialize() tworzone są obiekty, do których dostęp jest potrzebny w kontekście całej biblioteki, jest to odpowiednie miejsce na inicjalizację globalnych wskaźników.

W tym miejscu następuje uzyskanie uchwytu do obiektu typu ICorProfilerInfo2, który do udostępnia zestaw metod pozwalających na komunikacje ze środowiskiem CLR, umożliwiających jej monitorowanie i uzyskiwanie dodatkowych informacji o programie.

Kolejnym krokiem jest zarejestrowanie tego obiektu jako odbiorcy określonych zdarzeń określonych poniżej:

Listing 5.2: Zdarzenia

```
COR_PRF_MONITOR_NONE  = 0,
COR_PRF_MONITOR_FUNCTION_UNLOADS  = 0x1,
COR_PRF_MONITOR_CLASS_LOADS = 0x2,
COR_PRF_MONITOR_MODULE_LOADS  = 0x4,
COR_PRF_MONITOR_ASSEMBLY_LOADS  = 0x8,
COR_PRF_MONITOR_APPDOMAIN_LOADS = 0x10,
COR_PRF_MONITOR_JIT_COMPILATION = 0x20,
COR_PRF_MONITOR_EXCEPTIONS  = 0x40,
COR_PRF_MONITOR_GC  = 0x80,
COR_PRF_MONITOR_OBJECT_ALLOCATED  = 0x100,
COR_PRF_MONITOR_THREADS = 0x200,
COR_PRF_MONITOR_REMOTING  = 0x400,
COR_PRF_MONITOR_CODE_TRANSITIONS  = 0x800,
COR_PRF_MONITOR_ENTERLEAVE  = 0x1000,
COR_PRF_MONITOR_CCW = 0x2000,
COR_PRF_MONITOR_REMOTING_COOKIE = 0x4000 |
    COR_PRF_MONITOR_REMOTING,
COR_PRF_MONITOR_REMOTING_ASYNC  = 0x8000 |
    COR_PRF_MONITOR_REMOTING,
COR_PRF_MONITOR_SUSPENDS  = 0x10000,
COR_PRF_MONITOR_CACHE_SEARCHES  = 0x20000,
COR_PRF_MONITOR_CLR_EXCEPTIONS  = 0x1000000,
COR_PRF_MONITOR_ALL = 0x107ffff,
COR_PRF_ENABLE_REJIT  = 0x40000,
COR_PRF_ENABLE_INPROC_DEBUGGING = 0x80000,
COR_PRF_ENABLE_JIT_MAPS = 0x100000,
COR_PRF_DISABLE_INLINING  = 0x200000,
COR_PRF_DISABLE_OPTIMIZATIONS = 0x400000,
COR_PRF_ENABLE_OBJECT_ALLOCATED = 0x800000,
COR_PRF_ENABLE_FUNCTION_ARGS  = 0x2000000,
COR_PRF_ENABLE_FUNCTION_RETVAL  = 0x4000000,
COR_PRF_ENABLE_FRAME_INFO = 0x8000000,
COR_PRF_ENABLE_STACK_SNAPSHOT = 0x10000000,
COR_PRF_USE_PROFILE_IMAGES  = 0x20000000,
```

Do realizacji celów przedstawionych przed biblioteką AsProfiled potrzebne jest określenie następującej kombinacji flag:

```
COR_PRF_MONITOR_ENTERLEAVE |
COR_PRF_ENABLE_FUNCTION_RETVAL |
COR_PRF_ENABLE_FUNCTION_ARGS |
COR_PRF_ENABLE_FRAME_INFO
```

Pozwala to na otrzymywanie komunikatów na temat wejścia/wyjścia do/z metody wraz z danymi na temat jej argumentów i wartości zwracanej.

ICorProfiler

IMetaDataImport Interfejs spełnia kluczową rolę w procesie uzyskiwania informacji na temat dowolnych encji zdefiniowanych w ramach aplikacji przeznaczonych na platformę .NET. IMetaDataImport zawiera cały szereg metod, dzięki którym możliwe jest odszukanie i odczytanie wartości metadanych. Metoda dzielą się na cztery główne kategorie :

- Enumerating collections of items in the metadata scope.
- Finding an item that has a specific set of characteristics.
- Getting properties of a specified item.
- The Get methods are specifically designed to return single-valued properties of a metadata item. When the property is a reference to another item, a token for that item is returned. Any pointer input type can be NULL to indicate that the particular value is not being requested. To obtain properties that are essentially collection objects (for example, the collection of interfaces that a class implements), use the enumeration methods.

W bibliotece AsProfiled interfejs IMetaDataImport wykorzystywany jest do odczytu metadanych na temat kontraktów oraz metod nimi udekorowanymi. Informacje na temat metod uzyskiwane są przy pomocy wywołania funkcji

5.4 Odbieranie notyfikacji o zdarzeniach zachodzących w programie

W zadaniu ewaluacji kontraktów nakładanych na metody, kluczowe jest, aby biblioteka mogła odbierać zdarzenia na temat wywołania metody oraz wyjścia z niej. W tym celu niezbędne jest przekazanie informacji do maszyny CLR, na temat funkcji, które te zdarzenia będą obsługiwały. Cel ten realizowany jest poprzez wywołanie metody o sygnaturze

Listing 5.3: SetEnterLeaveFunctionHooks2

```
HRESULT SetEnterLeaveFunctionHooks2(
    [in] FunctionEnter2    *pFuncEnter,
    [in] FunctionLeave2    *pFuncLeave,
    [in] FunctionTailcall2 *pFuncTailcall);
```

na rzecz obiektu implementującego interfejs ICorProfilerInfo2. Jako argumenty podawane są wskaźniki do funkcji zdefiniowanej w ramach biblioteki AsProfiled. Zgodnie z dokumentacją MSDN, metody te muszą zostać udekorowane atrybutem __declspec(naked) co oznacza ze kompilator nie generuje dla tych funkcji tzw. prologu ani epilogu, czyli odpowiednich fragmentów kodu, które przywracają odpowiedni stan stosu oraz rejestrów. Po wykonaniu tych czynności możliwe jest przekazanie sterowania do innych funkcji, gdzie przetwarzanie zdarzenia jest kontynuowane. W ramach AsProfiled funkcje te są zadeklarowane w sposób następujący:

Listing 5.4: FunctionEnter

```
// deklaracja funkcji wywoływanej w momencie wejścia do metody
FunctionEnter(FunctionID functionID, UINT_PTR clientData,
    COR_PRF_FRAME_INFO func, COR_PRF_FUNCTION_ARGUMENT_INFO *
    argumentInfo);
```

oraz

Listing 5.5: FunctionLeave

```
// deklaracja funkcji wywoływanej w momencie wyjścia z metody
FunctionLeave(FunctionID functionID, UINT_PTR clientData,
    COR_PRF_FRAME_INFO func, COR_PRF_FUNCTION_ARGUMENT_RANGE *
    retvalRange);
```

,gdzie functionID - identyfikator funkcji, który może być użyty do uzyskania dostępu do jej metadanych
argumentInfo - wskaźnik do struktury COR_PRF_FUNCTION_ARGUMENT_INFO, która określa położenie argumentów funkcji w pamięci
retvalRange - wskaźnik do struktury COR_PRF_FUNCTION_ARGUMENT_RANGE, która określa położenie wyniku funkcji w pamięci

Te funkcje mają jasne przełożenie na zadanie ewaluacji kontraktów. W ramach funkcji FunctionEnter odbywa się sprawdzenie warunków początkowych, a implementacja FunctionLeave zawiera w sobie sprawdzenie warunków końcowych.

5.5 Odczyt metadanych

W tym podrozdziale opisano kolejny krok procesu ewaluacji kontraktu, jakim jest interpretacja metadanych w celu uzyskania informacji o wywoływanych metodach

i jej argumentach. Wywołanie każdej z metod kontrolowanego programu powoduje wywołanie metody FunctionEnter. W ramach tej funkcji odczytywane są informacje na temat metody, co do której otrzymano powiadomienie. Jest to możliwe dzięki otrzymywaniu jej identyfikatora w postaci argumentu functionID funkcji FunctionEnter, a następnie wykorzystaniu go do uzyskania dostępu do metadanych. Krok ten realizowany jest poprzez wywołanie metody o sygnaturze

Listing 5.6: GetTokenAndMetaDataFromFunction

```
HRESULT GetTokenAndMetaDataFromFunction(
    [in]  FunctionID functionId,
    [in]  REFIID     riid,
    [out] IUnknown   **ppImport,
    [out] mdToken    *pToken);
```

na rzecz obiektu implementującego interfejs ICorProfilerInfo2. Poprzez tą metodę uzyskujemy wartość typu mdToken, który jednoznacznie identyfikuje położenie informacji dotyczących funkcji. Niezbędne funkcje pozwalające na dostęp i interpretację metadanych metody zostały zgrupowane w obrębie klasy CMethodInfo, które udostępnia publiczny interfejs pozwalający na następujące operacje :

Listing 5.7: MethodInfo.h

```
WCHAR* GetMethodName();
CorCallingConvention GetCallingConvention();
ULONG GetArgumentsCount();
mdTypeDef GetTypeToken();
PCCOR_SIGNATURE GetMethodSignatureBlob();
mdMethodDef GetMethodToken();
CParam* GetReturnValue();
std::vector<CParam*>* GetArguments();
```

5.6 Parsowanie wyrażeń zawartych w kontraktach

Wyrażenia określające kontrakt muszą być zbudowane zgodnie z regułami gramatyki podanej w sekcji 5.2. Pierwszym krokiem na drodze do ich ewaluacji jest proces parsowania. W tym celu wykorzystano silnik Astudillo Visual C++ (odniesienie). Biblioteka mając zadane wyrażenie rozkłada ja na tokeny zdefiniowane w ramach gramatyki, a następnie tworzy drzewo rozbioru wyrażenia. Poniżej prezentowane jest efekt rozbioru kilku przykładowych wyrażeń:

Listing 5.8: Przykład1

```
"c.test.member == 31 && divided > 1"
Program
 Boolean Exp
```

```
 Cmp Exp
  Value
   Identifier:c.test.member
  Cmp Operator
   ==:==
  Value
   DecimalNumber:31
 Boolean Operator
  &&:&&
 Cmp Exp
  Value
   Identifier:divided
  Cmp Operator
   >:>
  Value
   DecimalNumber:1
```

Listing 5.9: Przykład2

```
"divided / divisor > 0 && @returnValue == 0 || val == \"test\""
Program
 Boolean Exp
  Boolean Exp
   Cmp Exp
    Mult Exp
     Value
      Identifier:divided
     Mult Operator
      /:/
     Value
      Identifier:divisor
    Cmp Operator
     >:>
    Value
     DecimalNumber:0
   Boolean Operator
    &&:&&
   Cmp Exp
    Value
     ReturnValue:@returnValue
    Cmp Operator
     ==:==
    Value
     DecimalNumber:0
  Boolean Operator
   ||:||
  Cmp Exp
   Value
    Identifier:val
   Cmp Operator
```

```
 ==:==
Value
 StringLiteral:"test"
```

5.7 Inspekcja wartości zmiennych

Jednym z etapów koniecznych w procesie wyliczania wartości wyrażeń zawartych w kontraktach jest wykonywanie podstawień wartości argumentów pod ich wystąpienia.

5.7.1 Typy proste

Bloby, odczytywanie wartosci

5.7.2 Typy złożone

rekrusywne przeszukiwanie typow

5.8 Ewaluacja kontraktów

Drzewo rozbioru

5.8.1 Ramki funkcji

5.8.2 Zachowywanie wartości początkowych

Rozdział 6

Porównanie z innymi bibliotekami

6.1 CodeContracts

6.2 LinFu.Contracts

Rozdział 7

Podsumowanie

Bibliografia