

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

Spis treści	i
1 Wstęp	1
1.1 Platforma .NET i CLR	1
1.2 Założenia	2
2 Programowanie kontraktowe	3
2.1 Historia	3
2.2 Opis	4
3 Profilowanie	7
3.1 Profilowanie – opis	7
3.2 Typy profilowania	8
3.2.1 Instrumentacja	8
3.2.2 Próbkowanie	8
3.3 Profilowanie w środowisku .NET	8
3.4 Profilowanie a dynamiczna weryfikacja programów	8
4 Omówienie funkcjonalności biblioteki	9
4.1 Inspekcja nadzorowanego programu	9
4.2 Kontrakty	10
4.3 Metadane	10
4.3.1 Reprezentacja metadanych	10
4.3.2 Interpretacja metadanych	14
4.4 Odczytywanie wartości obiektów	14
4.5 Wartości początkowe	15
4.6 Wartości zwracane	16
4.7 Ograniczenia	16
5 Szczegóły implementacji	19

5.1	Atrybuty jako kontrakty	19
5.2	Gramatyka kontraktów	20
5.3	Omówienie i implementacja interfejsów	21
5.4	Odbieranie notyfikacji o zdarzeniach zachodzących w programie	25
5.5	Odczyt metadanych	25
5.6	Inspekcja wartości zmiennych	25
5.6.1	Typy proste	25
5.6.2	Typy złożone	25
5.7	Parsowanie wyrażeń zawartych w kontraktach	25
5.8	Ewaluacja kontraktów	25
5.8.1	Ramki funkcji	25
5.8.2	Zachowywanie wartości początkowych	25
6	Porównanie z innymi bibliotekami	27
6.1	CodeContracts	27
6.2	LinFu.Contracts	27
7	Podsumowanie	29
	Bibliografia	31

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 pozafunkcyjnych, 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 programowania 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 implementacji tej koncepcji. Część 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:


```
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#:

```
[ 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 przysyłać 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.

One field in the CLI header is the RVA for the metadata directory, which gives access to all the metadata used by the assembly

Metadata stores declarative information about runtime types (classes, value types, and interfaces), global-functions and global-variable. Each such abstraction in a given metadata scope carries an identity as an mdToken (metadata token), where an mdToken is used by the metadata engine to index into a specific metadata data table in that scope. The metadata APIs return a token from each Define method and it is this token that, when passed into the appropriate Get method, is used to obtain its associated attributes.

A metadata token is a 4-byte value. The most significant byte (MSB) specifies the token type and consequently identifies the abstraction and its associated metadata table. For example, a value of 1 in the MSB means that the token is an mdTypeRef token, which represents a type reference, and that its metadata is stored in the TypeRef metadata table; a value of 4 in the MSB corresponds to an mdFieldDef token. The CorTokenType enumeration is used to specify the token types.

The lower three bytes, referred to as the record identifier (RID), contain the index of the row within the metadata table to which the token's MSB refers. For example, the metadata token with value 0x02000007 refers to row 7 in the TypeDef table in the current scope. Similarly, token 0x0400001A refers to row 26 (decimal) in the FieldDef table in the current scope. Row zero of a metadata table never contains data, so a metadata token whose RID is zero is referred to as a nil token. The metadata API defines a host of such nil tokens, one for each token type, such as mdTypeRefNil, with the value 0x01000000.

To be more concrete: a metadata token is a 4-byte value. The most-significant byte specifies what type of token this is. For example, a value of 1 means it's a TypeDef token, whilst a value of 4 means it's a FieldDef token. (For the full list, with their values, see the CorTokenType enumeration in CorHdr.h) The lower 3 bytes give the index of the row, within a MetaData table, that the token refers to. We call those lower 3 bytes the RID, or Record IDentifier. So, for example, the metadata token with value 0x01000007 is a 'shorthand' way to refer to row number 7 in the TypeDef table, in the current scope. Similarly, token 0x0400001A refers to row number 26 (decimal) in the FieldDef table in the current scope. We never store anything in row zero of a metadata table. So a metadata token, whose RID is zero, we call a "nil" token. The metadata API defines a host of such nil tokens – one for each token type (for example, mdTypeDefNil, with value 0x01000000).

The following table lists the metadata token types, the abstraction that each token type represents, and the name of the metadata table that contains the abstraction's metadata. All token types are variations of mdToken, which is the basic token type.

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.

mdIfacImpl

IfacImpl

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 naturalna 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 informacji zawartych w metadanych możliwy jest odczyt niezbędnych informacji. W opisywanej bibliotece zaimplementowano uzyskiwanie następujących danych:

Metody: `GetMethodName` `GetCallingConvention` `GetArgumentsCount` `GetReturnValue` `ReadArgumentsValues` `GetAttribute`

Parametry: `GetType`

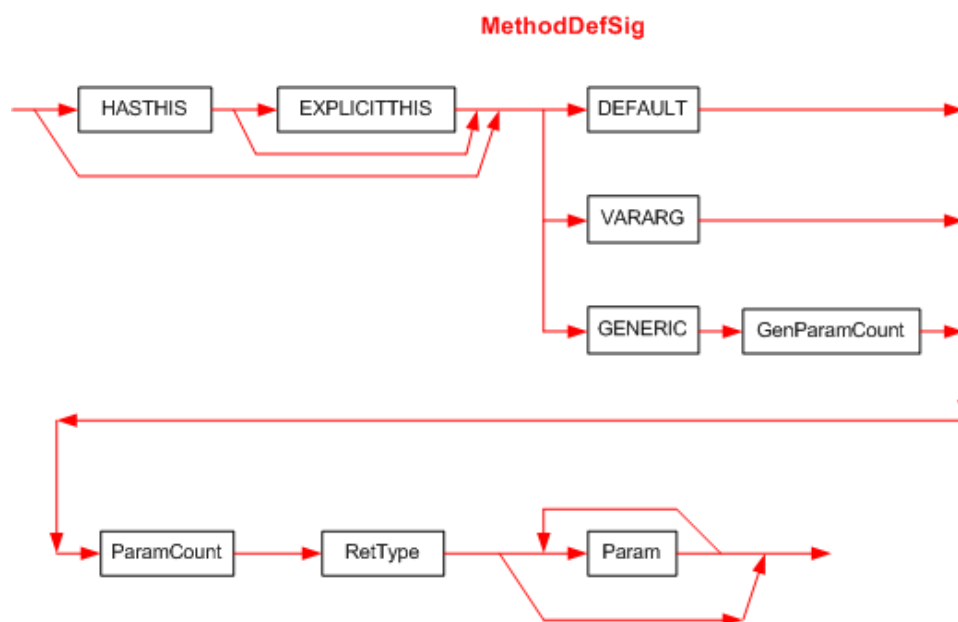
Typy: `GetName`

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 0xFFFFFFFF. 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 użyciu, 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 muszą być ewaluowalne do dwóch wartości: prawdy albo fałszu, co oznacza odpowiednio, że kontrakt jest lub też nie został spełniony.


```

|      '!= '

<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. ICorProfilerCallback2 zdefiniowany jest w następujący sposób:

```

// STARTUP/SHUTDOWN EVENTS
STDMETHOD(Initialize)(IUnknown *pICorProfilerInfoUnk);
STDMETHOD(Shutdown)();

```

```

// APPLICATION DOMAIN EVENTS
STDMETHOD(AppDomainCreationStarted)(AppDomainID appDomainID);
STDMETHOD(AppDomainCreationFinished)(AppDomainID appDomainID,
    HRESULT hrStatus);
STDMETHOD(AppDomainShutdownStarted)(AppDomainID appDomainID);
STDMETHOD(AppDomainShutdownFinished)(AppDomainID appDomainID,
    HRESULT hrStatus);
// ASSEMBLY EVENTS
STDMETHOD(AssemblyLoadStarted)(AssemblyID assemblyID);
STDMETHOD(AssemblyLoadFinished)(AssemblyID assemblyID, HRESULT
    hrStatus);
STDMETHOD(AssemblyUnloadStarted)(AssemblyID assemblyID);
STDMETHOD(AssemblyUnloadFinished)(AssemblyID assemblyID,
    HRESULT hrStatus);
// MODULE EVENTS
STDMETHOD(ModuleLoadStarted)(ModuleID moduleID);
STDMETHOD(ModuleLoadFinished)(ModuleID moduleID, HRESULT
    hrStatus);
STDMETHOD(ModuleUnloadStarted)(ModuleID moduleID);
STDMETHOD(ModuleUnloadFinished)(ModuleID moduleID, HRESULT
    hrStatus);
STDMETHOD(ModuleAttachedToAssembly)(ModuleID moduleID,
    AssemblyID assemblyID);
// CLASS EVENTS
STDMETHOD(ClassLoadStarted)(ClassID classID);
STDMETHOD(ClassLoadFinished)(ClassID classID, HRESULT hrStatus)
;
STDMETHOD(ClassUnloadStarted)(ClassID classID);
STDMETHOD(ClassUnloadFinished)(ClassID classID, HRESULT
    hrStatus);
STDMETHOD(FunctionUnloadStarted)(FunctionID functionID);
// JIT EVENTS
STDMETHOD(JITCompilationStarted)(FunctionID functionID, BOOL
    flsSafeToBlock);
STDMETHOD(JITCompilationFinished)(FunctionID functionID,
    HRESULT hrStatus, BOOL flsSafeToBlock);
STDMETHOD(JITCachedFunctionSearchStarted)(FunctionID functionID
    , BOOL *pbUseCachedFunction);
STDMETHOD(JITCachedFunctionSearchFinished)(FunctionID
    functionID, COR_PRF_JIT_CACHE result);
STDMETHOD(JITFunctionPitched)(FunctionID functionID);
STDMETHOD(JITInlining)(FunctionID callerID, FunctionID calleeID
    , BOOL *pfShouldInline);
// THREAD EVENTS
STDMETHOD(ThreadCreated)(ThreadID threadID);
STDMETHOD(ThreadDestroyed)(ThreadID threadID);

```

```

STDMETHOD( ThreadAssignedToOSThread )( ThreadID managedThreadID ,
    DWORD osThreadID );
// REMOTING EVENTS
// Client-side events
STDMETHOD( RemotingClientInvocationStarted ) ( );
STDMETHOD( RemotingClientSendingMessage )( GUID *pCookie , BOOL
    flsAsync );
STDMETHOD( RemotingClientReceivingReply )( GUID *pCookie , BOOL
    flsAsync );
STDMETHOD( RemotingClientInvocationFinished ) ( );
// Server-side events
STDMETHOD( RemotingServerReceivingMessage )( GUID *pCookie , BOOL
    flsAsync );
STDMETHOD( RemotingServerInvocationStarted ) ( );
STDMETHOD( RemotingServerInvocationReturned ) ( );
STDMETHOD( RemotingServerSendingReply )( GUID *pCookie , BOOL
    flsAsync );
// CONTEXT EVENTS
STDMETHOD( UnmanagedToManagedTransition )( FunctionID functionID ,
    COR_PRF_TRANSITION_REASON reason );
STDMETHOD( ManagedToUnmanagedTransition )( FunctionID functionID ,
    COR_PRF_TRANSITION_REASON reason );
// SUSPENSION EVENTS
STDMETHOD( RuntimeSuspendStarted )( COR_PRF_SUSPEND_REASON
    suspendReason );
STDMETHOD( RuntimeSuspendFinished ) ( );
STDMETHOD( RuntimeSuspendAborted ) ( );
STDMETHOD( RuntimeResumeStarted ) ( );
STDMETHOD( RuntimeResumeFinished ) ( );
STDMETHOD( RuntimeThreadSuspended )( ThreadID threadid );
STDMETHOD( RuntimeThreadResumed )( ThreadID threadid );
// GC EVENTS
STDMETHOD( MovedReferences )( ULONG cmovedObjectIDRanges , ObjectID
    oldObjectIDRangeStart [ ] , ObjectID newObjectIDRangeStart [ ] ,
    ULONG cObjectIDRangeLength [ ] );
STDMETHOD( ObjectAllocated )( ObjectID objectID , ClassID classID );
STDMETHOD( ObjectsAllocatedByClass )( ULONG classCount , ClassID
    classIDs [ ] , ULONG objects [ ] );
STDMETHOD( ObjectReferences )( ObjectID objectID , ClassID classID ,
    ULONG cObjectRefs , ObjectID objectRefIDs [ ] );
STDMETHOD( RootReferences )( ULONG cRootRefs , ObjectID rootRefIDs
    [ ] );
// EXCEPTION EVENTS
// Exception creation
STDMETHOD( ExceptionThrown )( ObjectID thrownObjectID );
// Search phase
STDMETHOD( ExceptionSearchFunctionEnter )( FunctionID functionID );

```

```

STDMETHOD(ExceptionSearchFunctionLeave)();
STDMETHOD(ExceptionSearchFilterEnter)(FunctionID functionID);
STDMETHOD(ExceptionSearchFilterLeave)();
STDMETHOD(ExceptionSearchCatcherFound)(FunctionID functionID);
STDMETHOD(ExceptionCLRCatcherFound)();
STDMETHOD(ExceptionCLRCatcherExecute)();
STDMETHOD(ExceptionOSHandlerEnter)(FunctionID functionID);
STDMETHOD(ExceptionOSHandlerLeave)(FunctionID functionID);
// Unwind phase
STDMETHOD(ExceptionUnwindFunctionEnter)(FunctionID functionID);
STDMETHOD(ExceptionUnwindFunctionLeave)();
STDMETHOD(ExceptionUnwindFinallyEnter)(FunctionID functionID);
STDMETHOD(ExceptionUnwindFinallyLeave)();
STDMETHOD(ExceptionCatcherEnter)(FunctionID functionID,
    ObjectID objectID);
STDMETHOD(ExceptionCatcherLeave)();
// COM CLASSIC VTable
STDMETHOD(COMClassicVTableCreated)(ClassID wrappedClassID,
    REFGUID implementedIID, void *pVTable, ULONG cSlots);
STDMETHOD(COMClassicVTableDestroyed)(ClassID wrappedClassID,
    REFGUID implementedIID, void *pVTable);
// End of ICorProfilerCallback interface implementation

// ICorProfilerCallback2 interface implementation
STDMETHOD(ThreadNameChanged)(ThreadID threadId, ULONG cchName,
    WCHAR name[]);
STDMETHOD(GarbageCollectionStarted)(int cGenerations, BOOL
    generationCollected[], COR_PRF_GC_REASON reason);
STDMETHOD(SurvivingReferences)(ULONG cSurvivingObjectIDRanges,
    ObjectID objectIDRangeStart[], ULONG cObjectIDRangeLength[])
    ;
STDMETHOD(GarbageCollectionFinished)();
STDMETHOD(FinalizeableObjectQueued)(DWORD finalizerFlags,
    ObjectID objectID);
STDMETHOD(RootReferences2)(ULONG cRootRefs, ObjectID rootRefIds
    [], COR_PRF_GC_ROOT_KIND rootKinds[], COR_PRF_GC_ROOT_FLAGS
    rootFlags[], UINT_PTR rootIds[]);
STDMETHOD(HandleCreated)(GCHandleID handleId, ObjectID
    initialObjectID);
STDMETHOD(HandleDestroyed)(GCHandleID handleId);
// End of ICorProfilerCallback2 interface implementation

```

ICorProfiler IMetaData

5.4 Odbieranie notyfikacji o zdarzeniach zachodzących w programie

5.5 Odczyt metadanych

5.6 Inspekcja wartości zmiennych

5.6.1 Typy proste

Bloby, odczytywanie wartości

5.6.2 Typy złożone

rekursywne przeszukiwanie typów

5.7 Parsowanie wyrażeń zawartych w kontraktach

AStudio

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
