

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 Omówienie funkcjonalności biblioteki	7
3.1 Inspekcja nadzorowanego programu	9
3.2 Kontrakty	9
3.3 Metadane	10
3.3.1 Reprezentacja metadanych	10
3.4 Odczytywanie wartości argumentów metod	12
3.5 Wartości zwracane	14
3.6 Wartości początkowe	15
4 Szczegóły implementacji	18
4.1 Atrybuty jako kontrakty	20
4.2 Gramatyka kontraktów	20
4.3 Omówienie i implementacja interfejsów	22
4.4 Odbieranie notyfikacji o zdarzeniach zachodzących w programie	25
4.5 Odczyt metadanych	26
4.5.1 Interpretacja metadanych	26
4.5.2 Odczyt metadanych w bibliotece AsProfiled	31
4.6 Parsowanie wyrażeń zawartych w kontraktach	32

4.7	Inspekcja wartości zmiennych	33
4.7.1	Typy wbudowane	35
4.7.2	Odwołania do wartości składowych obiektów złożonych	36
4.8	Ewaluacja kontraktów	37
4.8.1	Zachowywanie wartości początkowych	38
4.8.2	Wartości zwracane	38
5	Porównanie z innymi bibliotekami	40
5.1	CodeContracts	40
5.2	LinFu.Contracts	44
5.3	Porównanie funkcjonalności opisywanych bibliotek	51
6	Podsumowanie	52

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.

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. 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, aczkolwiek język C# jest uważany za flagowe rozwiązanie służące do tworzenia aplikacji pod tą platformę. Z drugiej strony, do tej pory powstało wiele innych rozwiązań, część z nich została zbudowana bezpośrednio przez firmę Microsoft, inne powstały jako niezależne projekty. Do tej pierwszej grupy, poza

wspomnianym C#, zaliczają się takie języki jak C++/CLI, J#, F#, Delphi 8 dla .NET, Visual Basic .NET. W drugiej grupie znajdują się m.in. Scala, Iron-Python, IronRuby, Nemerle. Z każdym z tych języków związany jest odpowiedni kompilator, którego zadaniem jest translacja programów 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. W niniejszej pracy wszystkie przykłady opierają się na programach napisanych w języku C#.

1.2 Założenia

W celu zapewnienia jak największej użyteczności, zostały przyjęte pewien zbiór założeń 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 poza funkcjonalnych, 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 podnoszenia ich jakości. Nie inaczej jest w przypadku programowania kontraktowego. Po raz pierwszy w obecnej postaci wprowadził je Bertrand Meyer w 1986 roku przy okazji 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
- języki oparte na platformie .NET w wersji 4.0

Drugą grupę stanowią języki dla których powstały nakładki umożliwiające ten rodzaj weryfikacji. Ta grupa jest znacznie bardziej 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 niezmiennik, to znaczy, warunek jaki musi być spełniony przed i po wywołaniu dowolnej publicznej metody w obrębie tej klasy.

Z kolei dla metod, kontrakt definiowany jest 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, tego rodzaju kontrakty opisywane są w sposób następujący:

Listing 2.1: Programowanie kontraktowe

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

Dla ilustracji, poniżej zamieszczono definicję klasy *ACCOUNT*, reprezentującej konto użytkownika w banku. W ramach konta można przeprowadzić operacje wpłaty i wypłaty środków. Dla metod odpowiadającym tym czynnościom, zdefiniowane zostały warunki początkowe i końcowe. Zgodnie z powyższym, warunki zostały umieszczone w ramach bloków *require* i *ensure*

Listing 2.2: Kontrakty w języku Eiffel

```
class ACCOUNT feature
  balance: INTEGER
  minimum_balance: INTEGER is 1000

  deposit (sum: INTEGER) is
    -- Zdeponowanie kwoty na koncie.
    require
      sum >= 0
    do
      add (sum)
    ensure
      balance = old balance + sum
    end

  withdraw (sum: INTEGER) is
    -- Wypłata pewnej sumy z konta.
    require
      sum >= 0
      sum <= balance - minimum_balance
    do
      add (-sum)
    ensure
      balance = old balance - sum
    end

  feature {NONE}
    add (sum: INTEGER) is
      -- Dodanie pewnej kwoty do sumy zdeponowanej na koncie.
```



```
do
    balance := balance + sum
end
end -- class ACCOUNT
```

Fragment ten nie wymaga szerszego komentarza, warto jednak zwrócić uwagę na słowo kluczowe *old* wykorzystane w warunkach końcowych. Jego wystąpienie przed zmienną oznacza odwołanie do wartości początkowej, czyli wartości na którą wskazywała ta zmienna przed rozpoczęciem wykonywania metody. Fakt, iż słowo to jest częścią języka Eiffel, pokazuje, że koncepcja programowania kontraktowego była wzięta pod uwagę już na etapie jego projektowania.

Sytuacja ma odmienny charakter w przypadku platformy .NET, gdzie nie występuje bezpośrednie wsparcie dla tego typu koncepcji. Przy budowie prezentowanej aplikacji wykorzystano cechę szczególną środowiska .NET, a w szczególności języka C#, jaką jest możliwość dekorowania metod atrybutami. Ten element języka będzie dokładniej opisany w dalszej części pracy, przy omawianiu funkcjonalności biblioteki. 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. Dla ilustracji, poniżej została zademonstrowana ogólna postać zapisu kontraktów jako atrybuty w języku C#:

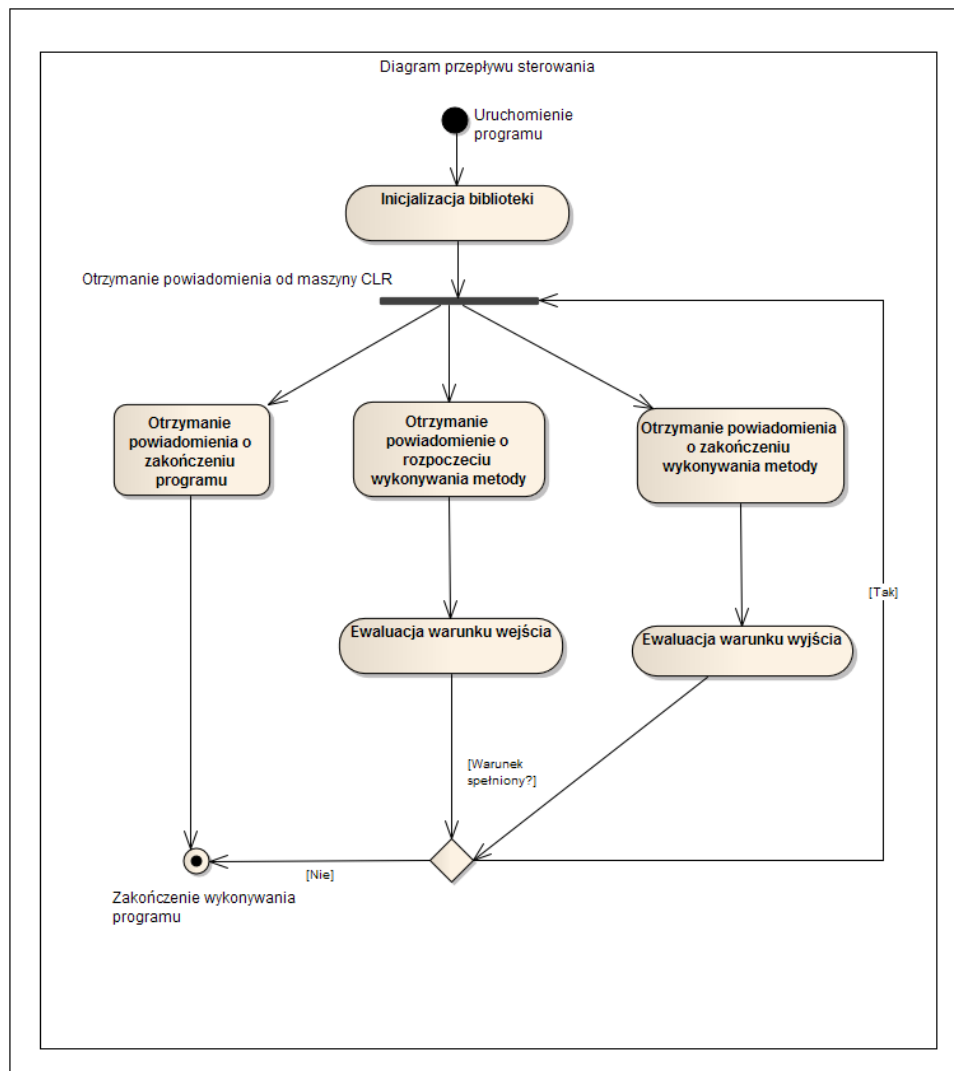
Listing 2.3: Definicja kontraktu

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

Rozdział 3

Omówienie funkcjonalności biblioteki

W tym rozdziale opisane zostały funkcjonalności jakie udostępnia biblioteka. Poniżej przedstawiony jest ogólny schemat działania aplikacji, którego poszczególne elementy rozpatrzone są w dalszej części.



Rysunek 3.1: Schemat działania aplikacji

3.1 Inspekcja nadzorowanego programu

Jak to zostało wspomniane we wcześniejszych rozdziałach, aplikacja weryfikująca kontrakty ma postać biblioteki COM. Niesie to ze sobą pewne konsekwencje. Przede wszystkim biblioteka musi być zgodna z konwencją definiowaną dla tego typu bibliotek. Szczegóły dotyczące tego tematu przedstawione zostały w kolejnych rozdziałach. Drugą konsekwencją tego jest konieczność rejestracji takiej biblioteki w systemie. Do tego celu używana jest aplikacja o nazwie `regsrv32.exe`, która to jest jednym z narzędzi dostarczanych wraz z platformą .NET. Pobiera ona identyfikator biblioteki, którym jest wartość typu GUID, a następnie umieszcza w rejestrze systemu klucz składający się z tego identyfikatora i ścieżki w systemie plików, określającej położenie biblioteki.

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. Odbywa się to w podany poniżej sposób:

Listing 3.1: Inicjalizacja środowiska

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

Powyższe zmienne są następnie odczytywane przez środowisko uruchomieniowe .NET. Informują one maszynę wirtualną, że ta powinna przesyłać informacje o zdarzeniach do biblioteki, której położenie określanie jest przy wykorzystaniu identyfikatora GUID.

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

3.2 Kontrakty

Podstawowym elementem, dzięki któremu możliwa jest weryfikacja metod, jest możliwość definiowania kontraktu. Zgodnie z tym co zostało powiedziane kontrakty definiujemy za pomocą atrybutów.

Atrybuty są to znaczniki o charakterze deklarycyjnym zawierające informację o elementach programu (np. klasach, typach wyliczeniowych, metodach) przeznaczoną do wykorzystania w trakcie działania programu.

Co jest w tym kontekście istotne to, iż są one pamiętane jako meta dane danego elementu programu. Definicja atrybutów jest jedynym elementem, wchodzącym bezpośrednio w skład omawianego rozwiązania, który musi znajdować się po stronie weryfikowanej aplikacji.

Atrybuty określające kontrakt mają postać:

Listing 3.2: Ogólna postać kontraktu

```
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 dotyczącym implementacji. Należy tu jednak wspomnieć, iż oba warunki zapisywane są jako łańcuchy znakowe. Oznacza to, iż przed ewaluacją takiego wyrażenia musi być zaimplementowany mechanizm pozwalający na jego analizę składniową. W jej wyniku otrzymywane jest drzewo rozbioru, które następnie poddawane jest ewaluacji. Ostatecznie, otrzymywana jest wartość określająca czy udekorowana metoda spełnia założony na nią kontrakt w kontekście danego wywołania.

3.3 Metadane

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

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

3.3.1 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 w nim uruchomiony. Aplikacje przeznaczone na platformę .NET naturalnie również muszą być zorganizowane w sposób zgodny z

tym standardem. Jednym z pól w ramach nagłówka PE jest offset określający położenie zbioru metadanych w ramach pliku wykonywalnego czy biblioteki.

W skład tego zbioru wchodzi pięć rodzajów strumieni. W tym kontekście, przez pojęcie strumień, określana jest sekcja w obrębie metadanych, w której przechowywane są informacje posegregowane względem ich typu. Te strumienie to:

1. `#Strings` - zawiera listę napisów, które określają nazwę programu, metod, parametrów.
2. `#US (User Strings)` - zawiera tablicę wszystkich stałych łańcuchowych definiowanych przez użytkownika.
3. `#GUID` - przechowuje listę wszystkich użytych w aplikacji 128 bitowych wartości GUID, m.in. tą która jednoznacznie identyfikuje aplikację
4. `#` - ten strumień jest tablicą tablic. Każda tablica nadrzędna identyfikowana przy pomocy pojedynczego bajtu o wartości od 0x00 do 0x29. Tablice wewnętrzne opisują metody, pola, parametry, typy.
5. `#Blob` - zawiera ciągi danych binarnych, które nie mogą być przedstawione w prosty sposób jako napisy. Dane te wykorzystywane są do opisywania sygnatur metod.

Metadane przechowują informacje na temat typów definiowanych w ramach programu (klasy, struktury, interfejsy), globalnych funkcji i zmiennych. Każda z tych abstrakcyjnych encji identyfikowana jest przez wartość typu `mdToken` (metadata token). Jest ona używana przez mechanizmy odczytujące metadane do określenia położenia w pamięci informacji na ich temat.

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 tablicy `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 typy tokenów wykorzystywane w ramach aplikacji, typy które opisują oraz nazwy tablic metadanych. Wszystkie tokeny są pochodnymi typu bazowego - mdToken.

Typ tokenu	Nazwa tablicy z metadanymi	Opis
mdTypeDef	TypeDef	Token odnoszący się do typów referencyjnych (klasy i interfejsy) i wartościowych (struktury)
mdMethodDef	MethodDef	Token odnoszący do informacji opisujących metody będące częścią klasy lub interfejsu
mdParamDef	ParamDef	Typ tokenu, którego wartości odnoszą się do informacji określających parametry metod
mdFieldDef	FieldDef	Odniesienie do metadanych dotyczących składowych wchodzących w skład klas, interfejsów
mdCustomAttribute	CustomAttribute	Zawiera odniesienie do metadanych atrybutów

3.4 Odczytywanie wartości argumentów metod

Dzięki informacjom zawartym w metadanych, możliwe jest określenie liczby i typów parametrów przekazywanych do metod. Są to dane statyczne, w tym sensie, że są one stałe w ramach raz zdefiniowanego programu. Z drugiej strony, wartości parametrów mogą być inne dla każdego wywołania metody, tak więc nie mogą w żaden sposób stanowić części metadanych. Musi więc zatem istnieć osobny mechanizm pozwalający na realizację tego zadania.

Okazuje się, że w momencie wysyłania powiadomienia o zajściu zdarzenia wywołania metody, maszyna wirtualna .NET wypełnia strukturę `_COR_PRF_FUNCTION_ARGUMENT_INFO`, która jest przekazywana do odbiorcy. Jej definicja przedstawia się w sposób następujący:

```
typedef struct _COR_PRF_FUNCTION_ARGUMENT_INFO {
    ULONG numRanges;
    ULONG totalArgumentSize;
    COR_PRF_FUNCTION_ARGUMENT_RANGE ranges[1];
} COR_PRF_FUNCTION_ARGUMENT_INFO;
```

Objaśnienia:

- *numRanges* - Liczba bloków pamięci, wewnątrz których znajdują się wartości argumentów. Określa liczbę elementów tablicy typu `COR_PRF_FUNCTION_ARGUMENT_RANGE`.
- *totalArgumentSize* - Całkowita długość wszystkich argumentów wyrażona w bajtach.
- *ranges* - Tablica obiektów typu `COR_PRF_FUNCTION_ARGUMENT_RANGE`, z których każda reprezentuje jeden blok pamięci, gdzie zawarte są wartości argumentów metody

Struktura ta reprezentuje wartości argumentów metody, zgodnie z porządkiem od lewej do prawej. Typ `COR_PRF_FUNCTION_ARGUMENT_RANGE` użyty w ramach tej struktury określony jest jak następuje:

```
typedef struct _COR_PRF_FUNCTION_ARGUMENT_RANGE {
    UINT_PTR startAddress;
    ULONG length;
} COR_PRF_FUNCTION_ARGUMENT_RANGE;
```

,gdzie

- *startAddress* - Adres początku bloku.
- *length* - Wielkość bloku pamięci.

Te dane w połączeniu z informacją o typie argumentu pozwalają na poprawny odczyt argumentów metody.

3.5 Wartości zwracane

Kolejnym elementem jest możliwość weryfikacji kontraktów zawierających w sobie odniesienia do wartości zwracanej przez metodę.

W dalszych przykładach wykorzystano klasę *Account* reprezentującą konto bankowe, której częściowa definicja zawarta jest poniżej.

Listing 3.3: Klasa *Account*

```
class Account {  
    // Identyfikator konta  
    public long AccountId;  
    // Stan środków na koncie  
    public int Balance;  
    // Nazwisko posiadacza  
    public string OwnerName;  
  
    ...  
}
```

Dodatkowo, niech dana będzie klasa *AccountManager* definiująca operacje na obiektach typu *Account*.

Listing 3.4: Klasa *AccountManager*

```
class AccountManager {  
    public Account ChangeOwnerName(Account account, string  
        newName) {  
        // Wywołanie metody rejestrującej bieżącą operację w  
        bazie danych  
        // Zwraca zapisaną wartość  
        account.OwnerName = Database.SetAccountOwnerInternal(  
            account.accountId, newName);  
        return account;  
    }  
}
```

Metoda ta zwraca obiekt typu *Account*, z uaktualnioną wartością *OwnerName* mówiącą o dostępnych środkach. W celu sprawdzenia poprawności działania tej funkcjonalności możliwe jest nałożenie odpowiedniego kontraktu na metodę *ChangeOwnerName*. Przy użyciu atrybutów z biblioteki *AsProfiled* możliwa jest inspekcja obiektu zwracanego z metody. Warto zaznaczyć, że takie odniesienia mają tylko sens przy warunkach końcowych, po odebraniu notyfikacji o zdarzeniu opuszczenia metody. W ramach atrybutów *AsContract* odwołanie do wartości

zwracanej odbywa się poprzez użycie identyfikatora *@returnValue*. Mając to na uwadze, warunek końcowy, sprawdzający czy nazwisko posiadacza rzeczywiście zostało uaktualnione, nałożony na tą metodę, mógłby mieć następującą postać:

Listing 3.5: Sprawdzenie poprawności operacji uaktualnienia nazwiska posiadacza konta

```
[AsContract(null, "@returnValue.OwnerName == newName")]
public Account ChangeOwnerName(Account account, string
    newName) {
    account.OwnerName = Database.SetAccountOwnerInternal(
        account.accountId, newName);
    return account;
}
```

Sposób odczytu wartości zwracanej jest analogiczny do odczytywania wartości parametrów metody. W tym wypadku, odbiorca zdarzenia otrzymuje dostęp do pojedynczego obiektu typu *COR_PRF_FUNCTION_ARGUMENT_RANGE*, który zawiera w sobie informacje na temat położenia wartości zwracanej w pamięci.

3.6 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 należy rozważyć jest przypadek, w którym odwołania do wartości początkowych ma miejsce w warunku końcowym w ramach danego kontraktu. 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 wywołania metody.

Niech dana będzie metoda zwiększająca bilans konta bankowego po wpłynięciu na nie środków.

Listing 3.6: Metoda zwiększająca bilans konta

```
class AccountManager {
    public Account Deposit(Account account, int sum) {
        // Wywołanie metody zapisującej do bazy danych nowy stan
        // konta o danym
        // identyfikatorze
        // Zwraca uaktualniony obiekt typu Account
        int newBalance = Database.DepositInternal(AccountId, sum
        );
        account.Balance = newBalance;
        return account;
    }
}
```

W tej sytuacji ma sens weryfikacja stanu konta po zakończeniu metody *Deposit*. W ramach biblioteki *AsProfiled* możliwe jest zdefiniowanie warunku końcowego, który weryfikuje tę wartość, przy uwzględnieniu wartości początkowej elementu *Balance*. W tej konkretnej sytuacji będzie miał on następującą postać:

Listing 3.7: Odwołanie do wartości początkowej

```
[AsContract(null, "account.Balance == ^account.Balance +
    sum")]
public Account Deposit(Account account, int sum) {
    int newBalance = Database.DepositInternal(AccountId, sum)
    ;
    account.Balance = newBalance;
    return account;
}
```

Kontrakt ten nakłada warunek określający, iż po spełnieniu po zakończeniu wykonywania metody nowa wartość pola *Balance* obiektu typu *Account* musi być równa początkowej wartości tego pola powiększonej o wartość *sum*. Odwołanie do wartości początkowych danego argumentu odbywa się poprzez poprzedzenie jego nazwy symbolem *^*, tutaj *^account.Balance*.

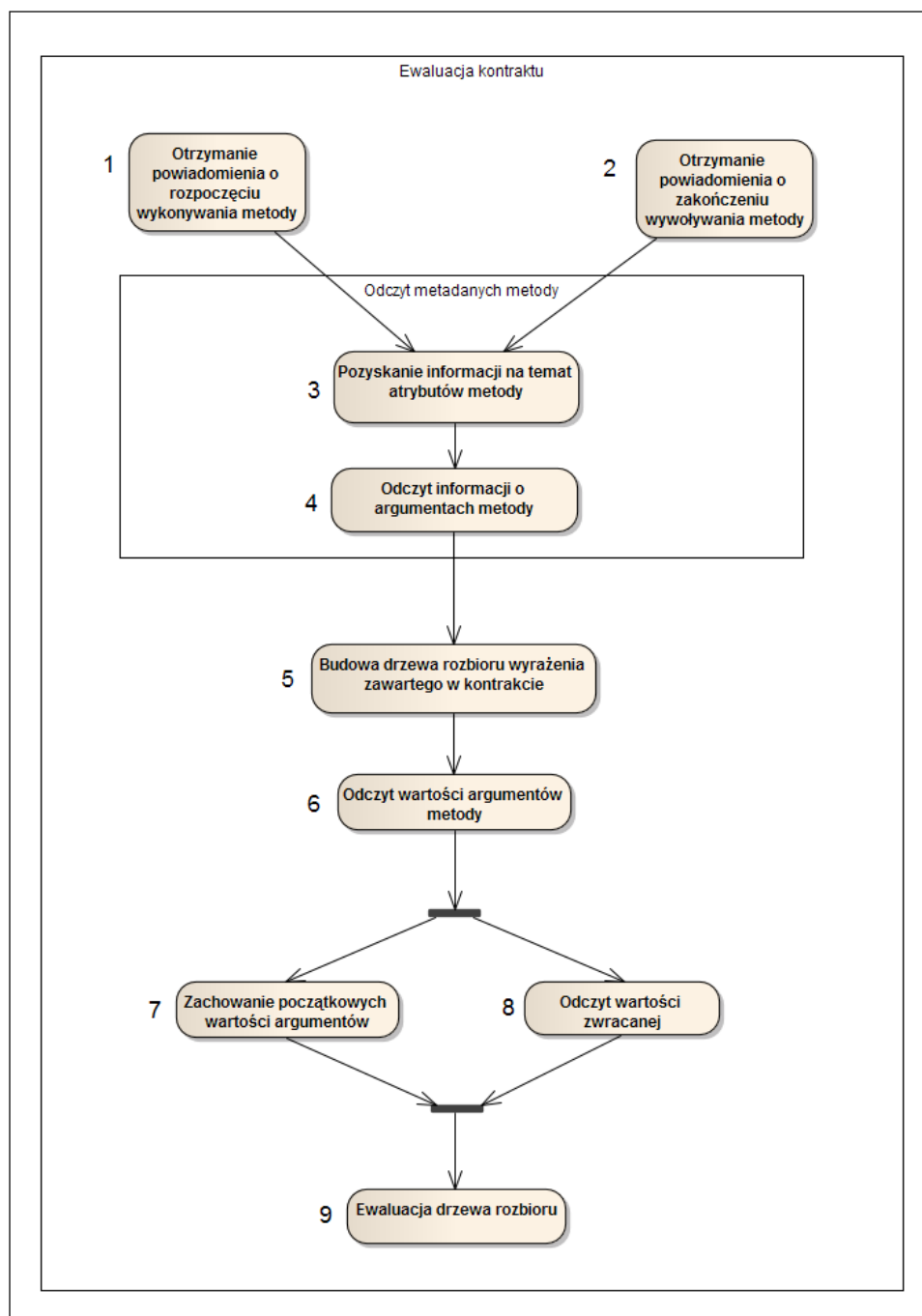
Zgodnie z tym co zostało powiedziane wcześniej, niezbędne jest zachowanie stanu obiektu w momencie otrzymania notyfikacji o rozpoczęciu wykonywania metody i odwoływanie się do niego w czasie weryfikacji warunków końcowych. Dodatkowo, nie wystarczy zachowanie referencji do obiektu. Wynika to z faktu, iż pomimo tego, że 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 stan obiektu 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 przetwarzania wstępnego, tzn. w chwili otrzymania powiadomienia o wywołaniu metody przeprowadzona zostaje analiza zarówno warunków początkowych i warunków końcowych kontraktu. W czasie tej analizy warunek końcowy sprawdzany jest pod kątem występowania elementów odnoszących się do stanu początkowego argumentów funkcji. Następnie argumenty, do których takie odwołania występują poddawane są inspekcji i 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 w momencie otrzymania powiadomienia o zakończeniu wykonywania weryfikowanej metody, kiedy to może zostać użyta do weryfikacji warunku końcowego.

Rozdział 4

Szczegóły implementacji

W tym rozdziale opisano szczegóły dotyczące implementacyjne biblioteki. Poniższy diagram ilustruje proces przetwarzania jakiemu poddawany jest kontrakt w celu oceny jego prawdziwości. Poszczególne elementy opisane są w odpowiadających im podrozdziałach.



Rysunek 4.1: Schemat przedstawiający proces ewaluacji kontraktów

4.1 Atrybuty jako kontrakty

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

Listing 4.1: Definicja atrybutu jako nośnika danych o kontrakcie

```
[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; }
}
```

Pierwsza linia określa, iż atrybut może być przypisywany tylko do metod oraz może występować co najwyżej jeden raz przy każdej z nich.

Centralnym elementem tej klasy jest dwuparametrowy konstruktor, przyjmujący dwa napisy jako parametry. Właśnie te napisy określają kontrakt. Pierwszy z nich określa warunek początkowy (*precondition*), a drugi dotyczy warunku końcowego (*postcondition*)

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

4.2 Gramatyka kontraktów

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

Listing 4.2: Gramatyka kontraktów

```
"Start Symbol" = <Program>
```

! Zbiory

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

! Symbole terminalne

```
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}
                  }*)*
```

! Symbole nieterminalne i zasady wyprowadzania wyrażeń

```
<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:

Listing 4.3: Przykładowe wyrażenia

```

i > 4 || test.inner == 3
i * j == k
str == "napis" || @returnValue == 0 && !false

```

Wyrażenia te nabierają sensu w momencie kiedy możliwe jest podstawienie wartości w miejsce identyfikatorów.

4.3 Omówienie i implementacja interfejsów

Wszystkie interfejsy przedstawione w tym i kolejnych rozdziałach zdefiniowane są w bibliotekach, wchodzący w skład SDK platformy .NET. W tabeli zaprezentowano wykorzystywane nagłówki wraz z opisem ich zawartości :

cor.h	Główny plik nagłówkowy zawierający API do operowania na metadanych
corhdr.h	Definicja struktur przechowujących metadane
corprof.h	Interfejsy profilujące

AsProfiled jak każda biblioteka typu COM udostępnia swoją funkcjonalność poprzez interfejsy. Niezależnie od przeznaczenia biblioteki musi ona przynajmniej implementować interfejs IUnknown, dzięki któremu możliwe jest uzyskanie uchwytu do innych obiektów implementujących bardziej szczegółowe interfejsy. 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. Na potrzeby tej pracy wystarczające jest omówienie dwóch z nich, mianowicie :

Listing 4.4: ICorProfilerCallback2

```
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 z listingu 4.4, jak sama nazwa wskazuje, są wywoływane podczas inicjalizacji biblioteki i w momencie zakończenia wykonywania programu. W ramach funkcji *Shutdown()* zwyczajowo zwalniane są uchwyt 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 do obiektów wykorzystywanych w trakcie działania aplikacji. W tym miejscu następuje też pozyskanie uchwytu do obiektu typu ICorProfilerInfo2, który to udostępnia zestaw metod pozwalających na komunikację ze środowiskiem CLR, umożliwiającą monitorowanie i uzyskiwanie dodatkowych informacji o programie.

Kolejnym krokiem jest zarejestrowanie tego obiektu jako odbiorcy podzbioru zdarzeń określonych poniżej.

Listing 4.5: 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.

Kolejnym wykorzystywanym interfejsem jest *IMetaDataImport*. Spełnia kluczową rolę w procesie uzyskiwania informacji na temat dowolnych encji zdefiniowanych w ramach aplikacji przeznaczonych na platformę .NET. *IMetaDa-*

talimport zawiera cały szereg metod, dzięki którym możliwe jest odszukanie i odczytanie wartości metadanych. Metoda dzieli się na cztery główne kategorie :

- Iterujące po kolekcjach zawierających encje z metadanymi opisującymi określoną encję.
- Odszukujące określoną encję na podstawie zadanych kryteriów.
- Pobierające informacje na temat konkretnych encji

W bibliotece *AsProfiled* interfejs *IMetaDataImport* wykorzystywany jest do odczytu metadanych na temat kontraktów oraz metod nimi udekorowanymi.

4.4 Odbieranie notyfikacji o zdarzeniach zachodzących w programie

W zadaniu ewaluacji kontraktów nakładanych na metody niezbędne jest aby biblioteka *AsProfiled* miała możliwość odbierania zdarzeń na temat wywołania metody oraz wyjścia z niej. Jest to realizowane poprzez przekazanie adresów funkcji zwrotnych, do maszyny CLR. Cel ten realizowany jest poprzez wywołanie metody o sygnaturze

Listing 4.6: *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 zdefiniowanych w ramach biblioteki *AsProfiled*. Zgodnie z dokumentacją MSDN, metody te muszą zostać udekorowane atrybutem `__declspec(naked)` co oznacza że kompilator nie generuje dla tych funkcji tzw. prologu ani epilogu, czyli odpowiednich fragmentów kodu, które przywracają odpowiedni stan stosu oraz rejestrów. Konsekwencją tego jest to, iż te czynności muszą być zaimplementowane w ramach biblioteki. Po ich wykonaniu możliwe jest przekazanie sterowania do innych funkcji, gdzie przetwarzanie zdarzenia jest kontynuowane. W ramach *AsProfiled* funkcje te są zadeklarowane w sposób następujący:

Listing 4.7: 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 4.8: 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, używany 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 bezpośrednie przełożenie na zadanie ewaluacji kontraktów. W ramach funkcji FunctionEnter odbywa się sprawdzenie warunków początkowych, a implementacja FunctionLeave zawiera w sobie sprawdzenie warunków końcowych.

4.5 Odczyt metadanych

W tym podrozdziale opisano kolejny krok na drodze do ewaluacji kontraktu, jakim jest interpretacja metadanych w celu uzyskania informacji o wywoływanych metodach i jej argumentach.

4.5.1 Interpretacja metadanych

Do zaimplementowania procesu weryfikacji programów przy obranym podejściu zachodzi potrzeba interpretacji metadanych. W tym celu potrzebna jest wiedza na temat reprezentacji metadanych i w jaki sposób należy je interpretować. Jak

to zostało opisane w poprzednim paragrafie, metadane przechowywane są w tablicach, do których dostęp uzyskiwany jest poprzez token odpowiedniego typu. W ramach tych tablic, które są częścią strumienia #~, znajdują się odwołania do strumienia #Blob. W strumieniu tym zawarte są dane opisujące sygnatury metod. Sygnatura opisująca metody zawiera w sobie wszelkie dotyczące jej informacje, m.in. typy argumentów, typ wartości zwracanej. Inne rodzaje sygnatur przechowują informację o typach pól klasy, propercji czy też zmiennych lokalnych. Niezależnie od opisywanego elementu języka, sygnatura przechowywana jest jako tablica bajtowa. Istnieje kilka rodzajów sygnatur, z których każda opisuje inny rodzaj encji:

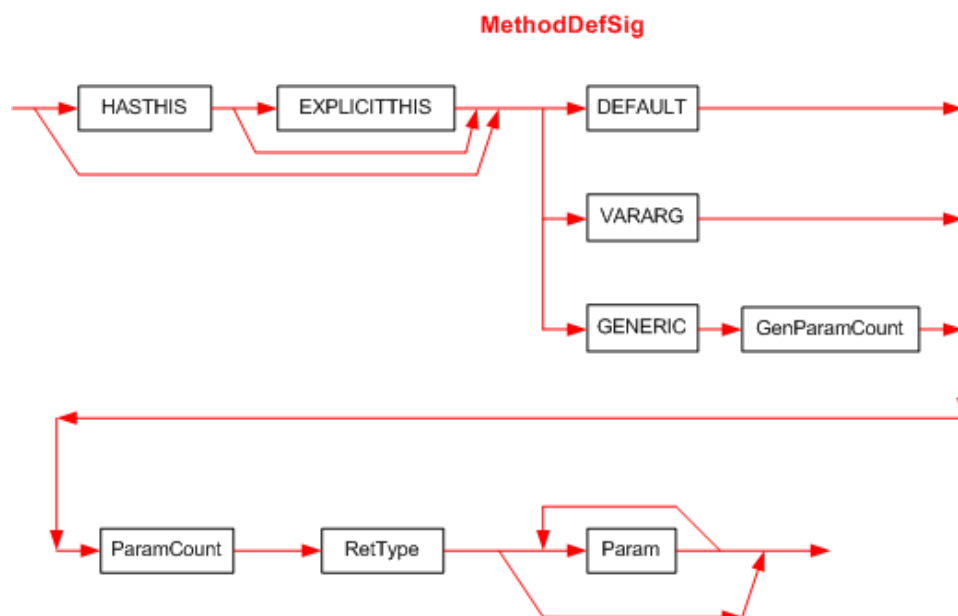
- MethodRefSig
- MethodDefSig
- FieldSig
- PropertySig
- LocalVarSig
- TypeSpec
- MethodSpec

W przypadku biblioteki AsProfiled konieczne jest odczytywanie informacji na temat metod, w tym celu konieczna jest funkcjonalność sygnatur typu MethodDefSig.

Poniżej przedstawiono strukturę tej sygnatury:

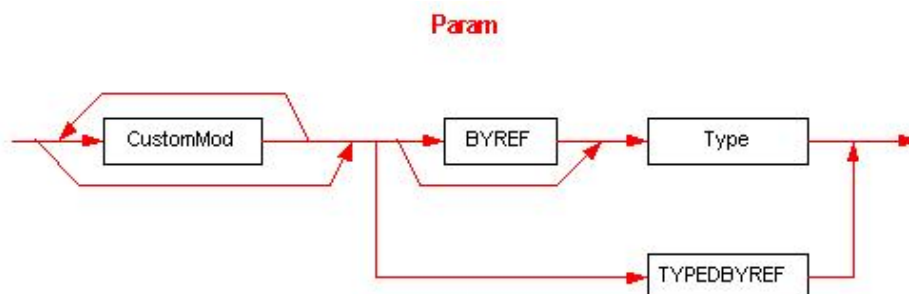
Legenda:

- HASTHIS = 0x20, EXPLICITTHIS = 0x40, DEFAULT = 0x0, VARARG = 0x5 - konwencja w jakiej wywoływana jest metoda
- GENERIC = 0x10 - oznaczenie określająca czy metoda posiada co najmniej jeden parametr generyczny
- GenParamCount - oznacza liczbę parametrów generycznych
- ParamCount - określa liczbę parametrów metody
- RetType - niesie informację o typie wartości zwracanej
- Param - opisuje typ każdego parametru metody, w ramach sygnatury element ten powinien występować ParamCount razy

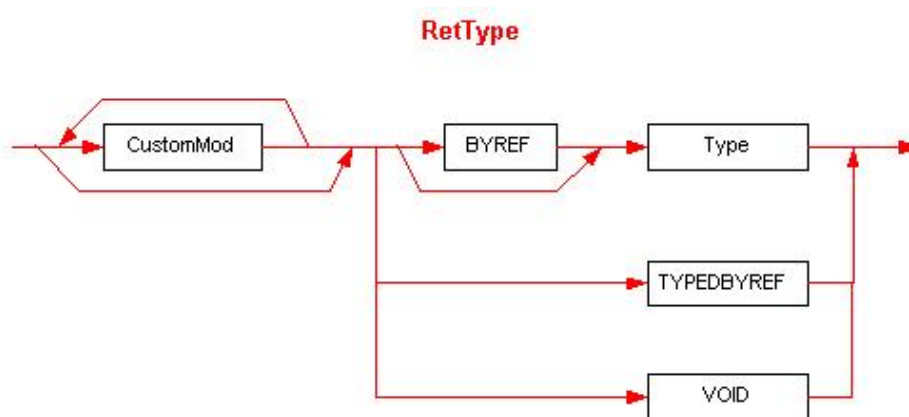


Rysunek 4.2: Struktura sygnatury metadanych opisujących metodę

Poniżej zamieszczono schematy ilustrujące strukturę parametrów metody oraz jej wartość zwracaną.



Rysunek 4.3: Struktura sygnatury parametru metody



Rysunek 4.4: Struktura sygnatury określającej wartość zwracaną

Powyższe ilustracje różnią się tylko dodatkowym rozgałęzieniem z elementem VOID, który oznacza, iż metoda nie zwraca żadnej wartości.

Element TYPE zdefiniowany jako jedna z wartości:

Listing 4.9: Znaczenie elementu TYPE

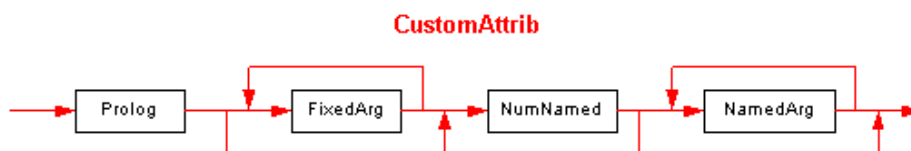
```

BOOLEAN | CHAR | I1 | U1 | I2 | U2 | I4 | U4 | I8 | U8 |
      R4 | R8 | I  | U  |
| VALUETYPE TypeDefOrRefEncoded
| CLASS TypeDefOrRefEncoded
| STRING
| OBJECT
| PTR CustomMod* VOID
| PTR CustomMod* Type
| FNPTR MethodDefSig
| FNPTR MethodRefSig
| ARRAY Type ArrayShape
| SZARRAY CustomMod* Type

```

Wartości te odpowiadają typom w ramach platformy .NET.

Innym ważnym elementem istotnym z punktu widzenia biblioteki AsProfiled jest struktura sygnatury atrybutów, które służą jako nośnik kontraktów. Ilustruje ją poniższy schemat:



Rysunek 4.5:

Interpretacja tych sygnatur jest podstawowym elementem, który należy wziąć pod uwagę w zadaniu ewaluacji kontraktów. Dzięki tym informacjom można określić typ obiektu, w szczególności argumentów funkcji, co z kolei umożliwia odczytanie ich aktualnych wartości w momencie wywołania funkcji.

Proces ten rozpoczyna się od pobrania wartości tokenu, opisanych w poprzedniej sekcji, odpowiedniego typu.

Tokeny odpowiedniego rodzaju uzyskiwane są poprzez wywołania odpowiednich metod na obiektach implementujących określone interfejsów. Dzięki informacją zawartych w metadanych możliwy jest odczyt niezbędnych informacji.

4.5.2 Odczyt metadanych w bibliotece AsProfiled

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 4.10: `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 4.11: Interfejs klasy *CMethodInfo*

```
WCHAR* GetMethodName(); // Odczytanie nazwy metody  
CorCallingConvention GetCallingConvention(); // Odczyt  
konwencji wywołania  
ULONG GetArgumentsCount(); // Pobieranie liczby argumentów  
mdTypeDef GetTypeToken(); // Pobranie wartości tokenu  
klasy, w której zawarta jest metoda  
%PCCOR_SIGNATURE GetMethodSignatureBlob(); // Adres do  
%mdMethodDef GetMethodToken(); //  
CParam* GetReturnValue(); // Pobiera informacje na temat  
wartości zwracanej  
std::vector<CParam*> GetArguments(); // Pobieranie  
informacji o argumentach funkcji
```

4.6 Parsowanie wyrażeń zawartych w kontraktach

Wyrażenia określające kontrakt muszą być zbudowane zgodnie z regułami gramatyki przedstawionej w 4.2. Pierwszym krokiem na drodze do ich ewaluacji jest proces analizy leksykalnej. W tym celu wykorzystano silnik Astudillo Visual C++. Biblioteka mając zadane wyrażenie rozkłada je na tokeny zdefiniowane w ramach gramatyki, a następnie tworzy drzewo rozbioru wyrażenia. Poniżej prezentowane jest efekt rozbioru kilku przykładowych wyrażeń:

Listing 4.12: Przykład 1

```
"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 4.13: Przykład 2

```
"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"

```

4.7 Inspekcja wartości zmiennych

Kolejnym etapem, koniecznym w procesie wyliczania wartości wyrażeń zawartych w kontraktach jest wykonywanie podstawień wartości argumentów pod ich wystąpienia.

Niech dane są następujące definicje klas:

Listing 4.14: Przykładowe klasy

```

class Test {
  public int member = 0;
}

class OtherClass {
  public Test test = new Test();
}

```

oraz metoda `TestMe`, na którą nałożono pewien kontrakt:

Listing 4.15: Kontrakt odwołujący się do parametrów metody

```
[AsContract("value > 1 && other.test.member == 31", null)]
public int TestMe(int value, OtherClass other)
{ }
```

Tak określony kontrakt definiuje warunek początkowy, po spełnieniu którego metoda `TestMe` może zostać wykonana. Wyrażenie zbudowane jest z dwóch warunków logicznych połączonych spójnikiem i (`&&`). Pierwszy warunek

`value > 1`

odnosi się do pierwszego argumentu funkcji, analogicznie, warunek

`other.test.member == 31`

odnosi się do drugiego parametru metody `TestMe(...)`. Elementem odróżniającym te dwa przypadki jest typ argumentu, do którego występuje odwołanie. Parametr *value* zalicza się do kategorii typów wartościowych wchodzących w skład języka, zaś parametr *other* jest typem referencyjnym, zdefiniowanym przez użytkownika. Niesie to ze sobą konsekwencje przy zadaniu odczytywania wartości argumentów.

Wszystkie typy są odwzorowanie na jedną z wartości wyliczeniowej *CorElementType* zdefiniowanej wewnątrz nagłówka *corHdr.h*. Poniższa tabela przedstawia częściową definicję typu wyliczeniowego, zawężoną do typów obsługiwanych przez bibliotekę *AsProfiled*.

Nazwa	Wartość	Opisywany typ
ELEMENT_TYPE_END	0x0	Niezdefiniowane
ELEMENT_TYPE_VOID	0x1	Typ zwracany void
ELEMENT_TYPE_BOOLEAN	0x2	Typ bool
ELEMENT_TYPE_CHAR	0x3	Wartość znakowy
ELEMENT_TYPE_I1	0x4	Typ short
ELEMENT_TYPE_U1	0x5	Typ short bez znaku
ELEMENT_TYPE_I2	0x6	Int
ELEMENT_TYPE_U2	0x7	Int bez znaku
ELEMENT_TYPE_I4	0x8	Long
ELEMENT_TYPE_U4	0x9	Long bez znaku
ELEMENT_TYPE_I8	0xA	Int64

ELEMENT_TYPE_U8	0xB	Int64 bez znaku
ELEMENT_TYPE_R4	0xC	Float
ELEMENT_TYPE_R8	0xD	Double
ELEMENT_TYPE_STRING	0xE	String
ELEMENT_TYPE_VALUETYPE	0x11	Typ wartościowy
ELEMENT_TYPE_CLASS	0x12	Typ referencyjny

Tabela 4.2: Typ wyliczeniowy

4.7.1 Typy wbudowane

W ramach platformy .NET definiowany pewien zbiór wbudowanych typów, które stanowią podstawę do definiowania wszystkich pozostałych. W kontekście tabeli 4.2 są to typy o wartościach mniejszych od 0x11. Poza typem *String*, który reprezentowany jest jako *ELEMENT_TYPE_STRING* wszystkie są typami wartościowymi. Fakt ten ma znaczenie w momencie odczytu wartości obiektów o takim typie.

Dla przypomnienia, w momencie wywołania metody po stronie weryfikowanego programu biblioteka AsProfiled otrzymuje powiadomienie o tym zdarzeniu. Wraz z nim przekazywana jest struktura *COR_PRF_FUNCTION_ARGUMENT_RANGE* (zob. 3.4), która zawiera adres do aktualnej wartości parametru. Dla typów wartościowych proces odczytania tej wartości polega na bezpośredniej interpretacji bajtów, których liczba określona jest przez pole *length* struktury *COR_PRF_FUNCTION_ARGUMENT_RANGE* znajdujących się pod adresem wskazywanym przez *startAddress*. W bibliotece AsProfiled zdefiniowana została klasa *ValueReader*, której odpowiedzialnością jest interpretowanie tych danych. Wszystkie metody, których zadaniem jest odczytanie wartości parametrów o typach prostych mają podobną konstrukcję. Przykład:

Listing 4.16: Odczytywanie wartości typu int

```
std::wstring CValueReader::TraceInt(UINT_PTR startAddress)
{
    std::wstringstream stream;
    stream << *(int *)startAddress;
    return stream.str();
}
```

Przy odczytywaniu wartości innych typów, zmianie ulega tylko linijka, w której odbywa się rzutowanie na odpowiedni typ.

Osobnego rozważenia wymaga odczytywanie parametrów typu *string*. W przeciwieństwie do wyżej opisanych wartości tego typu nie ma z góry określonej długości, i choćby z tego powodu nie jest możliwe jego odczytanie w sposób podany powyżej.

Interfejs *ICorProfilerInfo2* udostępnia w tym celu następującą metodę:

Listing 4.17: Odczytywanie wewnętrznej struktury napisów

```
HRESULT GetStringLayout (  
    [out] ULONG *pBufferLengthOffset,  
    [out] ULONG *pStringLengthOffset,  
    [out] ULONG *pBufferOffset)
```

W wyniku wywołania tej metody, pod przekazane wskaźniki, przypisywane są następujące wartości:

- *pBufferLengthOffset* - określa względne przesunięcie do adresu w pamięci, pod którym znajduje się wartość oznaczająca liczbę zarezerwowanych bajtów dla danego napisu
- *pStringLengthOffset* - względne przesunięcie do adresu, w którym określona jest rzeczywista długość napisu
- *pBufferOffset* - względne przesunięcie adresu, gdzie znajduje się pierwszy znak napisu

Wszystkie przesunięcia określają przesunięcie adresu pamięci w obrębie obiektu. Adres do miejsca w pamięci obiektu określony jest przez wartość elementu *startAddress* struktury *COR_PRF_FUNCTION_ARGUMENT_RANGE*. Mając te dane do dyspozycji, biblioteka *AsProfiled* jest w stanie odczytać parametrów typu *string*.

4.7.2 Odwołania do wartości składowych obiektów złożonych

Biblioteka *AsProfiled* pozwala definiować kontrakty, wewnątrz których znajdują się odwołania do pól złożonych struktur czy klas. Kontrakt zdefiniowany we fragmencie kodu 4.15 przedstawia przykładowe wyrażenie, w którym warunek

odnosi się do pola *member*, które jest składową klasy *Test*, a z kolei obiekt tego typu jest częścią klasy *other*. Ostatecznie, *member* jest typu prostego *int*, którego wartość odczytywana jest zgodnie z tym co zostało przedstawione w poprzednim punkcie. Pozostaje kwestia określenia miejsca w pamięci, w którym przechowywana jest ta wartość. W tym celu, biblioteka *AsProfiled* implementuje funkcjonalność inspekcji obiektów dowolnego typu, pod kątem zawierania składowych.

Niech wyrażenie jest postaci jak poprzednio *other.test.member > 31* Proces składa się z następujących kroków:

1. Przeszukanie listy parametrów metody w celu odnalezienie tego, do którego odwołanie znajduje się w kontrakcie. W tym przykładzie szukanym parametrem jest *other*.
2. Pobranie struktury opisującej ułożenie klasy, określającej parametr *other*
3. Przesunięcie wskaźnika na początek obiektu, zgodnie z informacjami zawartymi w strukturze pobranej w poprzednim kroku

Cały powyższy cykl powtarzany jest do momentu, aż odnalezione zostanie pole *member*, którego odczyt odbywa się już zgodnie z procedurą określoną w poprzedniej sekcji.

4.8 Ewaluacja kontraktów

Mając do dyspozycji opisane do tej pory funkcjonalności, biblioteka jest już w stanie wykonać zadanie ewaluacji kontraktu.

W tym celu utworzona została klasa *ClosureEvaluator*. Zawiera ona w sobie dwie metody publiczne, w ramach których ewaluowany jest kontrakt.

Listing 4.18: Interfejs klasy *ClosureEvaluator*

```
bool CClosureEvaluator::EvalPreCondition()
bool CClosureEvaluator::EvalPostCondition(
    COR_PRF_FUNCTION_ARGUMENT_RANGE *retvalRange)
```

Pierwsza z metod ewaluje warunek początkowy, druga warunek końcowy. Do sprawdzenia warunku końcowego potrzebna jest wartość zwracana z metody, którą to można odczytać wykorzystując argument *retvalRange*. Uwzględnienie wartości zwracanej jest jedynym elementem różniącym te metody. Poza tym ciąg wykonywanych czynności jest taki sam i przebieg zgodnie ze schematem:

1. Pobranie obiektu reprezentującego kontrakt
2. Utworzenie drzewa rozbioru wyrażenia opisującego kontrakt
3. Podmiana węzłów reprezentujących parametry na ich wartości
4. Ewaluacja drzewa

4.8.1 Zachowywanie wartości początkowych

W ramach wyrażeń opisujących kontrakty możliwe są odwołania do wartości parametrów początkowych, które zostały przekazane do metody w momencie jej wywołania. Wartości parametrów mogą ulec zmianie w czasie wykonywania metody, dlatego potrzebne jest traktowanie takich odwołań w specjalny sposób. Należy zauważyć, że takie odwołania mają tylko sens w przypadku wyrażeń określających warunek końcowy. Zgodnie z tym co zostało powiedziane wcześniej warunek ten ewaluowany jest po otrzymaniu powiadomienia o opuszczeniu metody, jednak wtedy nie jest już możliwe uzyskanie wartości początkowych parametrów, gdyż mogły one zostać zmienione w wyniku działań wewnątrz metody. Z tego względu konieczne jest ich skopiowanie w inny obszar pamięci, skąd będzie możliwe ich pobranie w dowolnym, późniejszym momencie. Odpowiednim momentem na przeprowadzenie tej operacji jest chwila, w której przychodzi powiadomienie o rozpoczęciu wykonywania kontrolowanej metody. Tą odpowiedzialność przejmuje obiekt typu *ClosureEvaluator*, dzięki informacją przekazywanym do konstruktora klasy.

Listing 4.19: Konstruktor klasy *ClosureEvaluator*

```
CCLclosureEvaluator::CCLclosureEvaluator(  
    CMethodInfo*, CAttributeInfo*,  
    ICorProfilerInfo2*, COR_PRF_FUNCTION_ARGUMENT_INFO*)
```

W ramach bloku inicjalizacyjnego dokonywana jest częściowa analiza warunku końcowego. Wyrażenie jest badane pod kątem występowania identyfikatorów oznaczających odwołanie do wartości początkowych. Po tym jak takowe zostaną odnalezione, następuje przeszukanie listy parametrów metody oraz odczytanie ich aktualnej wartości.

4.8.2 Wartości zwracane

Ostatnim elementem, które jest potrzebny do uzyskania w pełni funkcjonalnego mechanizmu ewaluacji kontraktów, a w szczególności warunków końcowych, jest mechanizm pozwalający na uwzględnianie i obsługę wartości zwracanych z metody. Zgodnie z tym co zostało napisane w rozdziale 3.5 wykorzystywana

jest tu struktura *COR_PRF_FUNCTION_ARGUMENT_RANGE*, która zawiera w sobie wskaźnik do miejsca w pamięci, w którym przechowywana jest aktualna wartość, która będzie zwrócony z metody. Odczyt jej nie powoduje żadnych komplikacji, gdyż wykorzystywane są tu dokładnie te same mechanizmy co przy interpretacji argumentów metody.

Rozdział 5

Porównanie z innymi bibliotekami

W tej części opisano dwa rozwiązania realizujące podejście programowania kontraktowego. Podobnie jak biblioteka *AsProfiled* są one przeznaczone dla programów pracujących w ramach platformy .NET.

5.1 CodeContracts

Jest to kompleksowe rozwiązanie firmy Microsoft, początkowe tworzone w ramach komórki badawczej Microsoft Research. Obecnie jest już dostępna na rynku w pełni funkcjonalna wersja tego zestawu narzędzi i wchodzi one w skład środowiska programistycznego Visual Studio 2010.

Z perspektywy implementacyjnej w ramach *Code Contracts* zastosowano zgoła inne podejście, od tego, które reprezentowane jest przez *AsProfiled*. Dla przypomnienia, *AsProfiled* stanowi zewnętrzną bibliotekę, do której odwołania występują na poziomie maszyny wirtualnej .NET. Sprawdzany program nie ma świadomości istnienia tej biblioteki, a jedynym elementem zewnętrznym jest klasa definiująca atrybuty, poprzez który reprezentowany jest kontrakt. W przypadku użycia *Code Contracts*, kontrolowany program (biblioteka) odwołuje się bezpośrednio do metod zdefiniowanych w ramach przestrzeni nazw *System.Diagnostics.Contracts*.

Razem z biblioteką dostarczane jest narzędzie *ccrewrite.exe*, które jest kluczowym elementem w zadaniu sprawdzania poprawności kontraktów w czasie działania aplikacji. Narzędzie to, dostając na wejściu skompilowaną bibliotekę, odnajduje wszelkie odwołania do metod zawartych w przestrzeni nazw *Sys-*

tem.Diagnostics.Contracts, a następnie modyfikuje jej kod pośredni. Transformacja ta polega na zamianie wszystkich odwołań do definicji kontraktów, na odpowiadające im bloki kodu. Na przykład, odwołania do wartości początkowych będą przetłumaczone na ciąg operacji w wyniku których zostaną one skopiowane i zachowane, dzięki czemu będą mogły być użyte w warunku końcowym.

Przepisywanie kodu skompilowanej biblioteki przy użyciu powyższego narzędzia to główna i zasadnicza różnica w podejściu w stosunku do tego co zostało wykorzystane w *AsProfiled*.

Narzędzie *Ccrewrite.exe* poprzez przepisywanie kodu umożliwia umieszczenie wszystkich deklaracji kontraktów na początku metody. Zmodyfikowany kod będzie zawierał sprawdzenie kontraktu końcowego we wszystkich miejscach, w których możliwe jest opuszczenie metody.

Z punktu widzenia użytkownika, zasadniczą różnicą, w stosunku do biblioteki *AsProfiled* jest sposób określania kontraktów. W tym wypadku są one definiowane jako wywołania statycznych metod klasy *Contract*, przy czym kontrakt podawany jako ich argument, w postaci wyrażenia logicznego. Poprawnym wyrażeniem jest dowolne wyrażenie zgodne z regułami ich budowania w ramach ustalonego języka programowania. Podejście to ma zasadniczą zaletę, mianowicie kontrakty są silnie typowane. Poniższy wycinek kodu ilustruje dotychczasowy opis:

Listing 5.1: *CodeContracts* - sposób użycia

```
public int Test(int arg1, int arg2)
{
    Contract.Requires(PreCondition);
    Contract.Ensures(PostCondition);
    int result = arg1 * arg2;
    return result;
}
```

Przy pomocy metod *Requires* i *Ensure* określane są odpowiednio warunki początkowe i warunki końcowe. Ich argumentem może być dowolnie skomplikowane wyrażenie obliczalne do wartości logicznej, a każda z tych metod może być wywołana dowolną liczbę razy.

Dla porównania, poniżej zestawiono deklaracje tego samego kontraktu początkowego w ramach bibliotek. *Code.Contracts* i *AsProfiled*:

Listing 5.2: Deklaracja kontraktu

```
// AsProfiled
[AsContract("arg1 != 0 && arg2 != 0", "@returnValue != 0")
]
public int Div(int arg1, int arg2)
{
    return arg1 / arg2;
}

// Code.Contracts
public int Div(int arg1, int arg2)
{
    Contract.Requires(arg1 != 0);
    Contract.Requires(arg2 != 0);
    Contract.Ensures(Contract.Result<int>() != 0);
    return arg1 / arg2;
}
```

W tym przykładzie, poza wcześniej wymienionymi metodami *Requires* i *Ensures*, znajduje się wywołanie metody *Contract.Results <int>()*. Jest to odpowiednik elementu *@returnValue* w obrębie kontraktów *AsContracts*, a więc oznacza odwołanie do wartości zwracanej z metody.

Kolejną wspólną cechą bibliotek *Code Contracts* oraz *AsProfiled* jest możliwość odwoływania się do wartości początkowych argumentów metody. W ramach kontraktów *AsContracts* odwołanie do wartości początkowych oznaczane są przez *^nazwaArgumentu*, z kolei tutaj tą rolę pełni metoda *Contracts.OldValue<T>(T value)*. Za zachowywanie tych wartości odpowiedzialny jest kod wygenerowany przez *ccewrite.exe*. Poniższy kod ilustruje wykorzystanie tej funkcjonalności.

Listing 5.3: Wykorzystanie wartości początkowych

```
public void Deposit(Account account, int amount)
{
    Contract.Requires(amount > 0);
    Contract.Ensures(account.Balance == Contract.OldValue(
        account.Balance) + amount);
    account.Balance += amount;
}
```

Odpowiednik powyższego kontraktu w ramach biblioteki *AsProfiled* jest następujący:

Listing 5.4: Wartości początkowe w *AsContract*

```
[AsContract("amount > 0", "~account.Balance + amount ==  
    account.Balance")]  
public void Deposit(Account account, int amount)  
{  
    account.Balance += amount;  
}
```

Dodatkowo *Code Contracts* udostępnia kilka innych, użytecznych funkcjonalności, których nie posiada biblioteka *AsProfiled*. Są to między innymi:

1. *Contract.Requires<TException>(bool condition)* - w przypadku, niespełnienia kontraktu, rzuca wyjątek określonego typu
2. *Contract.EnsuresOnThrow<TException>(bool condition)* - określa warunek końcowy jaki musi być zachowany w przypadku wystąpienia wyjątku typu *TException*
3. Deklaracja niezmienników, czyli warunków nakładanych na obiekt, które są sprawdzane przed i po wywołaniu każdej z publicznych metod obiektu.

Listing 5.5: Niezmiennik obiektu

```
[ContractInvariantMethod]  
private void ObjectInvariant ()  
{  
    Contract.Invariant ( condition1 );  
    Contract.Invariant ( condition2 );  
}
```

Metoda oznaczona atrybutem *[ContractInvariantMethod]* jest wywoływana przez odpowiedni kod wygenerowany przez narzędzie *ccrewrite.exe*

Podsumowując, *Code Contracts* w obecnej postaci jest dopracowanym rozwiązaniem, o czym świadczy fakt, iż jest częścią oficjalnej dystrybucji platformy .NET 4.0. Dużą zaletą w stosunku do biblioteki *AsProfiled*, poza bogatszą funkcjonalnością, jest silne typowanie wyrażeń określających warunki. Ułatwia to ich pisanie oraz uniemożliwia tworzenie niepoprawnych w sensie składniowym warunków.

5.2 LinFu.Contracts

LinFu.Contracts wchodzi w skład bibliotek zgrupowanych w ramach projektu LinFu, stworzonego przez Philipa Laureano a udostępnianego na zasadach wolnego oprogramowania. U podstaw tej biblioteki leży mechanizm generowania dynamicznych obiektów pośredniczących (ang. *dynamic proxy*), które pozwalają na przechwytywanie wywołań metod na docelowym obiekcie. Ten właśnie mechanizm wykorzystywany jest przy realizacji podejścia programowania kontraktowego.

Proces tworzenia obiektu pośredniczącego odbywa się poprzez metodę *CreateProxy<T>(IInvokeWrapper wrapper)* należącej do klasy *ProxyFactory*. W rezultacie zwracany jest uchwyt to obiektu typu *T*, przy czym każde odwołanie do jego metod będzie przechwytywane przez obiekt implementujący interfejs *IInvokeWrapper*. Interfejs ten określony jest w sposób następujący:

Listing 5.6: Interfejs IInvokeWrapper

```
public interface IInvokeWrapper
{
    void BeforeInvoke(InvocationInfo info);
    object DoInvoke(InvocationInfo info);
    void AfterInvoke(InvocationInfo info, object returnValue)
    ;
}
```

Powyższe metody wywoływane są przez obiekt pośredniczący w momentach zgodnych z ich nazewnictwem, tzn. *BeforeInvoke*, *AfterInvoke* oznaczają moment przed i po wywołaniu, natomiast *DoInvoke* to właściwe wywołanie. Należy zaznaczyć iż metoda na obiekcie docelowym może, ale nie musi być wykonana, decyduje o tym implementacja metody *DoInvoke*. Dla jasności prezentowany jest poniższy przykład:

Listing 5.7: Tworzenie obiektu pośredniego

```
// Klasa początkowa
public class Worker
{
    public virtual void Do()
    {
        Console.WriteLine("Metoda Do()");
    }
}

public class WrappedWorker : IInvokeWrapper
```

```
{
    object _target;
    public WrappedWorker(object target)
    {
        _target = target;
    }
    void BeforeInvoke(InvocationInfo info)
    {
        Console.WriteLine("Przed wywołaniem metody Do()");
    }

    object DoInvoke(InvocationInfo info)
    {
        Console.WriteLine("Wywołanie metody Do()");
        object result = null;
        // W celu wywołania oryginalnej metody Do() należy
        // usunąć komentarz z następnego wiersza:
        // result = info.TargetMethod.Invoke(_target, info.
        //     Arguments);
        return result;
    }

    void AfterInvoke(InvocationInfo info, object returnValue)
    {
        Console.WriteLine("Po wywołaniu metody Do()");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        ProxyFactory factory = new ProxyFactory();
        WrappedWorker worker = new WrappedWorker(new Worker());
        Worker wrappedWorker = factory.CreateProxy<Worker>(
            worker);

        worker.Do();
        // W rezultacie wyświetlone zostaną komunikaty:
        // Przed wywołaniem metody Do()
        // Wywołanie metody Do()
        // Po wywołaniu metody Do()
    }
}
```

Wiedząc czym jest obiekt pośredniczący, możliwe jest opisanie sposobu realizacji programowania kontraktowego przez bibliotekę *LinFu.Contracts*. Zasadniczo jest to rozwinięcie powyższej koncepcji. Wewnątrz przestrzeni nazw *LinFu.DesignByContract2.Contracts* znajduje się klasa *AdHocContract*, które zdefiniowana jest jak następuje:

Listing 5.8: Klasa *AdHocContract*

```
public class AdHocContract : IMethodContract,
    IContractProvider, ITypeContract
{
    public AdHocContract();
    public IList<IInvariant> Invariants { get; }
    public IList<IPostcondition> Postconditions { get; }
    public IList<IPrecondition> Preconditions { get; }
}
```

W jej polach przechowywane są odniesienia do inwariantów, warunków początkowych i warunków końcowych. W połączeniu z klasą typu *ContractChecker* implementującą interfejs *IInvokeWrapper* oraz mechanizmem tworzenia obiektów pośrednich oddawana jest funkcjonalność sprawdzania poprawności kontraktów.

Listing 5.9: *LinFu.Contracts*

```
// Obiekt, którego metody będą sprawdzane pod kątem
// poprawności
LinFuCalculator calculator = new LinFuCalculator();
AdHocContract contract = new AdHocContract();

ProxyFactory factory = new ProxyFactory();
ContractChecker checker = new ContractChecker(contract);
checker.Target = calculator;

// Obiekt pośredniczący
LinFuCalculator wrapped = factory.CreateProxy<
    LinFuCalculator>(checker);
// Każde wywołanie metody na rzecz tego obiektu będzie
// powodowało
// sprawdzenie odpowiednich kontraktów
wrapped.Mult(2, 3);
```

Powyższy fragment obrazuje sposób w jaki uaktywniany jest proces weryfikacji kontraktów, jednak w żaden sposób ich nie definiuje. W tym celu, konieczne jest ich zdefiniowanie, a następnie dołączenie ich do odpowiedniej listy zdefinio-

wanej w ramach klasy *AdHocContract*.

Wszystkie rodzaje warunków definiuje się poprzez implementację jednego z następujących interfejsów *Invariant*, *IPrecondition*, *IPostcondition*. Każdy z powyższych rozszerza interfejs *ICheckContract* zdefiniowany w następujący sposób:

```
public interface IContractCheck
{
    // określa czy dany kontrakt powinien być sprawdzony w
    // kontekście
    // aktualnie wywoływanej metody
    bool AppliesTo(object target, InvocationInfo info);
    // Reakcja na wyjątek wyrzucany przez błędnie
    // zdefiniowany kontrakt
    void Catch(Exception ex);
}
```

Warunki początkowe określa się poprzez utworzenie klasy implementującej interfejs *IPrecondition*:

```
public interface IPrecondition : IMethodContractCheck,
    IContractCheck
{
    bool Check(object target, InvocationInfo info);
    void ShowError(TextWriter output, object target,
        InvocationInfo info);
}
```

Metoda *Check* zawiera właściwą definicję kontraktu, w rezultacie zwracając wartość logiczną mówiącą o jego spełnieniu w kontekście wywoływanej metody. Pobiera ona dwa argumenty, pierwszy z nich jest obiektem na rzecz którego metoda została wywołana, drugi z nich zawiera informacje o tym wywołaniu, w szczególności zawiera wartości argumentów przekazanych do metody.

Mając te informacje możliwe jest stworzenie klasy określającej przykładowy warunek początkowy. Niech dana będzie klasa *Calculator* z metodą *Div(int arg1, int arg2)*. Definicja warunku początkowego w ramach *LinFu.Contracts* mogłaby mieć następującą postać:

Listing 5.10: Warunek początkowy w *LinFu.Contracts*

```
class CalculatorPrecondition : IPrecondition
{
```

```
public bool Check(object target, LinFu.DynamicProxy.
    InvocationInfo info)
{
    int argument1 = (int) info.Arguments[0];
    int argument2 = (int) info.Arguments[1];
    // warunek dla którego kontrakt nie jest spełniony
    if (argument1 == 0 || argument2 == 0)
        return false;
    // w pozostałych przypadkach warunek początkowy uważany
    // jest za spełniony
    return false;
}

public bool AppliesTo(object target, LinFu.DynamicProxy.
    InvocationInfo info)
{
    // Określenie, iż ten warunek ma mieć zastosowanie
    // tylko do klasy typu Calculator
    Calculator calculator = target as Calculator
    if (calculator == null)
        return false;
    // Sprawdzanie kontraktu ma się odbywać tylko dla metody
    // Div
    if (info.TargetMethod.Name == "Div")
        return true;
    return false;
}
...
}
```

Dla przypomnienia, pokazano jak ten sam kontrakt określany jest przy wykorzystaniu biblioteki *AsProfiled* i *Code.Contracts*

Listing 5.11: Deklaracja kontraktu

```
// AsProfiled
[AsContract("arg1 != 0 && arg2 != 0", null)]
public int Div(int arg1, int arg2)
{
    return arg1 / arg2;
}

// Code.Contracts
public int Div(int arg1, int arg2)
{
    Contract.Requires(arg1 != 0);
    Contract.Requires(arg2 != 0);
}
```

```
    return arg1 / arg2;  
}
```

Widoczna jest tu znaczna różnica w łatwości definiowania kontraktu oraz ilości kodu potrzebnego do uzyskania tego samego efektu.

Analogiczna sytuacja występuje dla warunków końcowych. Ponownie, konieczne jest zdefiniowanie klasy implementującej tym razem interfejs *IPostcondition*, który różni się od *IPrecondition* metodą *BeforeMethodCall* oraz zmienioną sygnaturą metody *Check*.

Listing 5.12: Deklaracja *BeforeMethodCall*

```
void BeforeMethodCall(object target, InvocationInfo info);  
bool Check(object target, InvocationInfo info, object  
    returnValue);
```

Metoda *Check* posiada dodatkowy argument *returnValue*, który oznacza wartość zwracaną metody (odpowiednik *@returnValue* z *AsProfiled* oraz *Contract.OldValue* z *CodeContracts*).

Metoda *BeforeMethodCall* została dodana w celu obsługi wartości początkowych argumentów i wykonywana jest przed wejściem do metody docelowej. W tym rozwiązaniu można dostrzec pewną analogię z podejściem użytym w *AsProfiled*. Tam też dokonywane jest przetwarzanie wstępne przed wejściem do metody, właśnie w celu przechowania stanu parametrów użytych w kontrakcie. W przeciwieństwie jednak do niej, tu za poprawne zachowanie ich wartości odpowiedzialny jest twórca kontraktu. W *AsProfiled* i *CodeContracts* czynność ta przerzucona została na bibliotekę i zachodzi całkowicie bez wiedzy osoby nakładającej więzy poprawności na metodę.

Mając wiedzę na temat definiowania warunków końcowych, możliwe jest porównanie ich wykorzystania w stosunku do tego co zostało pokazane przykładzie 5.4.

Listing 5.13: Klasa *Account*

```
public class Account  
{  
    public int Balance;  
    public virtual void Deposit(int amount)  
    {  
        this.Balance += amount;  
    }  
}
```

```
}  
}
```

Listing 5.14: Definicja warunku końcowego

```
class AccountPC : IPostcondition  
{  
    private int oldBalance;  
    public void BeforeMethodCall(object target,  
        InvocationInfo info)  
    {  
        oldBalance = (target as Account).Balance;  
    }  
  
    public bool Check(object target, InvocationInfo info,  
        object returnValue)  
    {  
        Account account = target as Account;  
        return account.Balance == oldBalance + (int)info.  
            Arguments[0];  
    }  
  
    public bool AppliesTo(object target, InvocationInfo info)  
    {  
        if (target as Account == null)  
            return false;  
        if (info.TargetMethod.Name == "Deposit")  
            return true;  
        return false;  
    }  
}
```

Warunek zawiera sprawdzenie bilans rachunku po wpłynięciu depozytu jest równy jego wartości początkowej powiększonej o zadaną wartość.

Podsumowując, biblioteka LinFu.Contracts spełnia wszystkie funkcjonalności niezbędne do programowania kontraktowego. Z drugiej strony można podejrzewać, iż nie było to głównym założeniem podczas projektowania LinFu. Definiowanie kontraktów wymaga dosyć dużego nakładu pracy w porównaniu do prezentowanych wcześniej rozwiązań.

Do zalet tego rozwiązania można z pewnością zaliczyć nieinwazyjność kontraktów w stosunku do sprawdzanego kodu. Zestaw kontraktów może być zawarty w osobnej bibliotece, załączanej w zależności od potrzeby. Z drugiej strony wymusza definiowanie klas jako publicznych, zaś metody które mają być prze-

słonięte muszą być oznaczone jako wirtualne - wynika to ze specyfiki koncepcji tworzenia dynamicznych obiektów pośredniczących.

5.3 Porównanie funkcjonalności opisywanych bibliotek

W tym miejscu prezentowane jest podsumowanie funkcjonalności przedstawionych bibliotek.

Cecha	AsProfiled	CodeContracts	LinFuContracts
Określenie kontraktu	Atrybut nakładany na metodę	Wewnątrz metody przy użyciu metod z biblioteki	Osobna klasa implementująca odpowiednie interfejsy
Użyte podejście	Zewnętrzna biblioteka Odbieranie powiadomień od maszyny CLR Interpretacja danych binarnych	Przepisywanie języka pośredniego skompilowanego programu	Dynamiczna generacja obiektu pośredniczącego
Warunki początkowe	✓	✓ (Contract.Requires)	✓ (interfejs IPrecondition)
Warunki końcowe	✓	✓ (Contract.Ensures)	✓ (interfejs IPostcondition)
Wartości początkowe	✓ (^identyfikator)	✓ (Contract.Old)	✓ (metoda BeforeMethodCall, wymaga ręcznej implementacji)
Wartość zwracana	✓ (@returnValue)	✓ (Contract.Result)	✓ (parametr returnValue)
Inwariant klasy	- (łatwa implementacja, wystarczy sprawdzać kontrakt dla klasy w momencie wywołania i zakończenia metody)	✓ (metoda oznaczona atrybutem [ContractInvariantMethod])	✓ (interfejs IInvariant)

Rozdział 6

Podsumowanie

Celem tej pracy była prezentacja rozwiązania zadania weryfikacji kontraktów, którego wyróżniającym się elementem jest wykorzystanie podejścia stosowanego przy budowie aplikacji profilujących. Okazuje się, że jest to pierwsza tego typu implementacja. Z pewnością zaletą takiego podejścia jest nieinwazyjność w kod wynikowy docelowej aplikacji.

Możliwym rozwinięciem funkcjonalności biblioteki byłoby uwzględnienie czasów wykonywania metody w ramach kontraktów, dodatkowo uzależniając je od aktualnych wartości przekazywanych do metod. Dodatkowo, w bardzo prosty sposób, mierząc czasy wykonywania metod, możliwe jest uzyskanie w pełni funkcjonalnej aplikacji profilującej, co razem z możliwością weryfikacji kontraktów daje użyteczne narzędzie pozwalające na poprawę jakości aplikacji docelowej.

Z drugiej strony wykorzystane mechanizmy nie udostępniają w prosty sposób funkcjonalności wykonywania metod w ramach weryfikowanego programu, co w pewien sposób ogranicza siłę wyrażen zawartych w kontraktach. Inną istotną sprawą jest pewna trudność, która występuje przy pracy z niskopoziomowymi interfejsami rodzaju *ICorProfileInfo*, *IMetaDataImport*, która powodowana jest brakiem obszerniejszej dokumentacji na ich temat.

Podsumowując, biblioteka *AsProfiled* stanowi ciekawy przykład realizacji programowania kontraktowego, a sama w sobie jest obszernym źródłem wiedzy na temat niskopoziomowej struktury programów na platformę .NET, ich reprezentacji w pamięci oraz możliwości komunikacji z maszyną .NET.