



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

**DYNAMICKÉ ANALYZÁTORY PRO PLATFORMU
SEARCHBESTIE**

DYNAMIC ANALYZERS FOR SEARCHBESTIE PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN JANOUŠEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2017

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací dynamického analyzátoru kontraktů s parametry. V první části práce je představena problematika testování paralelních programů, včetně metod testování a chyb, které se mohou v těchto programech nacházet. Podrobněji se zabývá metodou dynamické analýzy a věnuje se konkrétním dynamickým analyzátorům, jako jsou FastTrack nebo analyzátor kontraktů. Ve druhé části práce je popsán návrh a implementace dynamického analyzátoru kontraktů pro framework RoadRunner a platformu Searchbestie.

Abstract

This master thesis deals with the design and implementation of dynamic Contract analyzer with parameters. In the first part of the thesis the probematics of testing parallel programs are discussed, including methods of testing and errors, which can be found in parallel programs. It deals with dynamic analysis in detail and particular dynamic analyzers, such as FastTrack or Contract analyzer. The second part of the thesis also presents the design and implementation of dynamic Contract analyzer for RoadRunner framework and SearchBestie platform.

Klíčová slova

Dynamická analýza, testování paralelních programů, analyzátor kontraktů, SearchBestie, RoadRunner

Keywords

Dynamic analysis, testing of parallel programs, contract analyzer, SearchBestie, RoadRunner

Citace

JANOUŠEK, Martin. *Dynamické analyzátoru pro platformu SearchBestie*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Smrčka Aleš.

Dynamické analyzátoru pro platformu SearchBestie

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Janoušek
18. května 2017

Poděkování

Rád bych poděkoval mému vedoucímu Ing. Aleši Smrčkovi Ph.D. za odborné vedení, za pomoc a rady při zpracování této práce.

Obsah

1	Úvod	3
2	Testování vícevláknových programů	4
2.1	Souběžnost a relace Happens before	4
2.2	Logický čas	6
2.3	Synchronizační nástroje	6
2.4	Chyby v synchronizaci	7
2.4.1	Uváznutí (deadlock)	7
2.4.2	Blokování	8
2.4.3	Stárnutí	8
2.4.4	Časově závislá chyba nad daty	8
2.4.5	Porušení atomicity	9
2.4.6	Porušení kontraktu	9
2.5	Metody testování vícevláknových programů	9
2.5.1	Statická analýza	9
2.5.2	Dynamická analýza	9
2.5.3	Deterministické testování	10
2.5.4	Zátěžové testování	10
2.5.5	Vkládání šumu	10
2.6	Nástroje pro testování Java programů	11
2.6.1	RoadRunner	11
2.6.2	SearchBestie	13
2.6.3	Propojení SearchBestie a frameworku Roadrunner	14
3	Dynamické analyzátorы	15
3.1	Analyzátor FastTrack	15
3.1.1	Zápis následovaný čtením nebo zápisem	16
3.1.2	Čtení následované zápisem	16
3.2	Analyzátor kontraktů	16
3.2.1	Definice základního kontraktu	16
3.2.2	Rozšíření kontraktu	17
3.2.3	Dynamická analýza založena na LockSet algoritmu	18
3.2.4	Dynamická analýza založena na relaci Happens before	18
4	Návrh analyzátoru kontraktů	20
4.1	Specifikace požadavků	20
4.2	Přehled částí systému	21
4.3	Nástroj v Roadrunneru	21

4.3.1	Vnitřní reprezentace	23
4.3.2	Práce s šablonami a instancemi	24
4.3.3	Vytvoření nové instance	24
4.3.4	Krok advance	25
4.3.5	Vektorový čas	27
4.3.6	Informace o detekci porušení	28
4.4	Podpora nástroje v SearchBestie	28
4.4.1	Cover producer a úprava nástroje RRPlugin	28
4.4.2	ContractRICreator a úprava SearchBestie	29
5	Implementace analyzátoru kontraktů	33
5.1	Dynamický analyzátor kontraktů	33
5.1.1	Spuštění	33
5.2	Propojení analyzátoru se SearchBestie	34
5.2.1	Spuštění	34
5.3	Implementační problémy	34
5.3.1	Rozšíření RoadRunneru o parametry metod	35
5.3.2	Rozšíření RoadRunneru o návratový parametr	38
5.3.3	Ověření rozšíření	39
6	Ověření funkčnosti	40
6.1	Jednotkové testování	40
6.1.1	Zpracování konfiguračního souboru	40
6.1.2	Porovnání metod	41
6.2	Testování dynamického analyzátoru ve frameworku RoadRunner	41
6.2.1	Textový popis testů	41
6.2.2	Sekvenční diagram	42
6.3	Testování propojení analyzátoru kontraktů s platformou SearchBestie	43
7	Závěr	45
Literatura		46
Seznam příloh		47
A Návod na instalaci a spuštění		48
A.1	Instalace Roadrunneru	48
A.2	Instalace SearchBestie	48
B Testy funkčnosti dynamického analyzátoru		50
B.1	Testování jednoduchých kontraktů	50
B.2	Testování složitějších kontraktů	54
B.3	Testování kontraktů s parametry	66

Kapitola 1

Úvod

Testování softwaru je důležitou součástí jeho vývoje. Nalezení chyby ovšem v některých případech nebývá jednoduché, zvláště pak, pokud se jedná o chyby ve vícevláknových programech. Tyto chyby se mohou projevit pouze v určitých situacích a pro jejich testování musí být použito speciálních technik. Jedním z typů chyb, které se v těchto aplikacích vyskytují, je porušení kontraktů. Touto problematikou se zabývá mimo jiné výzkumná skupina VeriFIT, která vytvořila dynamický analyzátor detekující porušení kontraktů v programech napsaných v jazyce C/C++. Tato práce se oproti tomu zabývá dynamickou analýzou vícevláknových Java programů a její součástí je vytvoření dynamického analyzátoru, který slouží pro detekci porušení kontraktů v programech napsaných v jazyce Java, což dosud nebylo možné.

Před návrhem a implementací tohoto analyzátoru kontraktů je nutné představit problematiku testování vícevláknových programů, popsat různé metody tohoto testování, představit dostupné nástroje umožňující toto testování a v neposlední řadě ukázat problémy, které se mohou ve vícevláknových programech vyskytovat. Těmito uvedenými tématy se zabývá kapitola 2, ve které se také nachází popis frameworku RoadRunner a SearchBestie, pro které je analyzátor kontraktů vytvořen.

Následující kapitola 3 se zabývá dynamickými analyzátoři, přičemž detailněji je zde popsán analyzátor *FastTrack* a analyzátor kontraktů. Pro analyzátor kontraktů jsou zde představena jeho možná rozšíření, neboť tato rozšíření jsou součástí implementovaného řešení, a také se zde nachází způsoby detekce porušení kontraktů za pomoci dynamické analýzy.

Kapitola 4 obsahuje specifikaci požadavků a návrh samotného analyzátoru kontraktů, který zahrnuje především popis samotného nástroje, ale také návrh propojení s platformou SearchBestie. Nachází se zde definice formátu konfigurační souboru konaktu, popis a komunikace mezi částmi systému a popis činnosti analyzátoru.

V kapitole 5 jsou popsány implementační detaily a problémy, které se během implementace vyskytly a musely být řešeny. Další důležitou informací nacházející se v této kapitole je popis spuštění analyzátoru, a to jak samostatně, tak z platformy SearchBestie. Dále lze v této kapitole nalézt umístění souborů, které byly při implementaci vytvořeny nebo modifikovány jak ve frameworku RoadRunner, tak v platformě SearchBestie.

Poslední kapitola obsahuje popis testů, kterými byla ověřena funkčnost navrženého řešení. Pro tento účel bylo navrženo 46 testů, které jsou kvůli jejich množství přesunuty do přílohy B, a tak se v této kapitole nachází pouze popis jejich formátu. Dále se v této kapitole nachází popis jednotkových testů a ověření vlivu použití SearchBestie s tímto analyzátem.

Kapitola 2

Testování vícevláknových programů

V dnešní době nalezneme v běžných počítačích procesory, které mají 2 nebo více jader. Aby jeden paralelní program mohl využívat více těchto jader současně, musí využívat jeden z následujících způsobů. Prvním způsobem je použití více procesů. V tomto kontextu lze proces chápat jako sekvenčně prováděný samostatný program ve vlastním adresovém prostoru. [22, 1].

Druhým způsobem je použití *vícevláknového* programu, ve kterém se nachází pouze jeden proces obsahující několik vláken¹. V tomto kontextu proces chápeme jako obálku sady souběžně prováděných vláken v jednom adresovém prostoru. Přestože použití vláken přináší řadu výhod, jako například vyšší rychlosť přepínání kontextu a sdílení prostředků, přináší také problémy v podobě souběžného přístupu k těmto sdíleným prostředkům. Tento přístup musí být řízen pomocí tzv. *synchronizačních prostředků*, jinak může docházet k chybám způsobených paralelizací [1].

Základním požadavkem dnešních operačních systémů² je souběžné provádění více programů. Vzhledem k tomu, že počet procesů a vláken, které mají běžet souběžně, je většinou vyšší než počet fyzických jader počítače, musí být výpočetní čas procesoru rozprostřen mezi všechny tyto procesy a vlákna. Rozprostření výpočetního času zajišťuje *plánovač*, který plánuje přepínání kontextu procesů a vláken. Plánovač je typicky preemptivní³ a zohledňuje několik vlastností procesů a vláken, jako je například priorita, čas strávený na procesu, doba čekání nebo paměťové požadavky. Díky plánovači a dalším zdrojům nedeterminismu (obsluha přerušení, blokující operace, atd.) je paralelní provádění programu také nedeterministické, tj. procesy a vlákna mohou být různě *proloženy*. Tento nedeterminismus způsobuje, že chyby vzniklé paralelním prováděním programu se v jednom běhu mohou vyskytnout, zatímco v jiném ne. Z hlediska testování paralelních programů je tedy důležité ověřit, zda se chyba nevyskytuje v žádném možném proložení [18].

2.1 Souběžnost a relace Happens before

V paralelním programu je důležité určit pořadí jednotlivých událostí napříč procesy (vlákny). Protože výpočet jednotlivých procesů a vláken probíhá asynchronně, musí být zaveden způsob, jak pořadí těchto událostí určit. Vztah mezi dvěma událostmi, které se provedly ve stejném, nebo různém procesu (vlákně), se nazývá *relace kauzálního uspořádání* nebo taky relace *Happens before* [13].

¹V jazyce Java je program chápán jako proces s vláknem, přestože lze vytvořit i programy, které budou obsahovat více procesů. Nicméně každý proces obsahuje minimálně jedno vlákno [17].

²Dnešními operačními systémy jsou myšleny multiprogramové operační systémy, kde uživatel požaduje souběžný běh více programů, a tedy souběžný běh více procesů a vláken.

³Při preemptivním plánování může být procesor procesu odebrán bez jeho přičinění (ukončení procesu, zahájení čekání, atd.).

Relace Happens before byla představena Lesliem Lamportem pro prostředí distribuovaných systémů, a proto zde bude uvedena v tomto obecnějším kontextu. Nicméně tato relace platí jak pro uspořádání událostí napříč vlákny, tak i procesy. V této kapitole tedy budou uvažovány procesy a vlákna za totožné. Následující vysvětlení této relace vychází ze zdrojů [13, 10].

Paralelní program se skládá z množiny n asynchronních procesů p_1, p_2, \dots, p_n . Procesy spolu navzájem mohou komunikovat pouze pomocí zasílání zpráv⁴. Dále platí, že procesy nemohou sdílet svůj globální čas⁵ a že výpočet procesů, včetně zasílání zpráv, probíhá asynchronně⁶. Zprávu zaslannou procesem p_i procesu p_j označme jako m_{ij} . Vykonávání procesu se skládá ze sekvence atomických událostí, které mohou být tří typů [10]:

- interní akce – mění vnitřní stav procesu,
- odeslání zprávy m ($send(m)$) – mění vnitřní stav odesírajícího procesu,
- přijetí zprávy m ($recv(m)$) – mění vnitřní stav přijímajícího procesu.

Dále nechť e_i^x označuje x -tou událost procesu i . Pak platí, že události v procesu jsou lineárně uspořádány podle indexu x . Toto lineární uspořádání událostí procesu i označme jako relaci \rightarrow_i a množinu všech akcí procesu p_i označme jako h_i . Relace \rightarrow_i značí kauzální závislost nad procesem p_i , tj. zápis $e_i^x \rightarrow_i e_i^y$ značí, že událost e_i^x se stala před událostí e_i^y v procesu p_i . Dále definujme relaci \rightarrow_{msg} , která značí kauzální závislost mezi dvojicí událostí $send(m)$ a $recv(m)$.

Nechť $H = \bigcup_{i \in n} h_i$ značí množinu všech událostí vykonalých v paralelním programu. Pak *relace kauzálního uspořádání* (tj. relace **Happens Before**) \rightarrow je definována následovně [10]:

$$\forall e_i^x, \forall e_j^y \in H, e_i^x \rightarrow e_j^y \Leftrightarrow \begin{cases} e_i^x \rightarrow e_j^y, & \text{pokud } i = j \wedge x < y \\ \text{nebo} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{nebo} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{cases} \quad (2.1)$$

Výše definovaná relace *Happens before* umožňuje definovat uspořádání mezi dvěma událostmi. Pokud pro dvě události e_i a e_j platí $e_i \rightarrow e_j$, pak můžeme říct, že událost e_i předchází událost e_j . To znamená, že v systému nemůže nastat situace, kdy by se událost e_j vyskytla před událostí e_i . Pokud pro dvě události platí $e_i \not\rightarrow e_j$, pak událost e_i nepředchází událost e_j . Pro definované vztahy platí následující pravidla [10]:

$$e_i \not\rightarrow e_j \Rightarrow e_j \not\rightarrow e_i \quad (2.2)$$

$$e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i \quad (2.3)$$

Posledním možným vztahem dvou událostí je *souběžnost* (concurrency). Dvě události jsou souběžné, pokud platí [10]:

$$e_i \not\rightarrow e_j \wedge e_j \not\rightarrow e_i \quad (2.4)$$

Tento vztah se značí jako $e_i \parallel e_j$. Z hlediska odhalování chyb v paralelních programech je tento vztah nejdůležitější. Pokud jsou dvě události souběžné (tj. konkurentní), pak nelze říci, která

⁴Pod zasíláním zpráv je možné si představit také komunikaci pomocí sdílené paměti, přístupu k zámku, atd. Pro jednoduchost bude veškerá tato komunikace zobecněna jako zasílání zpráv mezi procesy.

⁵Globální čas procesu je pohled daného procesu na celkový čas (tj. čas všech procesů) v paralelním programu. Každý proces může mít ve stejnou chvíli jiný pohled na celkový čas.

⁶Předpokládejme, že každý proces běží na vlastním procesoru (jádře)

událost proběhne dříve. Může se stát, že při jednom spuštění bude nejprve provedena událost e_i a poté událost e_j , ale při jiném spuštění bude jejich pořadí opačné. Pokud budeme uvažovat reálný příklad, pak je tento problém mimo jiné v souběžném zápisu více vláken do jedné sdílené proměnné (viz kapitola 2.4).

2.2 Logický čas

V předchozí kapitole bylo definováno uspořádání událostí v jednotlivých procesech. Každé takové události je přiřazeno časové razítko, které značí čas procesu v době provádění této události. Nejedná se ovšem o běžný fyzický čas, ale o čas *logický* [13]. Logický čas je dostačující k určení uspořádání událostí a není nijak závislý na fyzickém čase.

Systém logického času se skládá z časové domény T a logických hodin C , přičemž na množině T je definována relace částečného uspořádání $<$, která není ničím jiným, než výše definovanou relací Happens before. Logické hodiny jsou funkce, která mapuje události e z množiny všech událostí H na prvky z množiny T ($C : H \rightarrow T$). Pomocí této funkce je možné určit relaci Happens before mezi dvěma událostmi e_i a e_j následovně:

$$e_i \rightarrow e_j \implies C(e_i) < C(e_j) \quad (2.5)$$

Logický čas může být implementován následujícími způsoby [10]:

- *Skalárni čas* (Scalar time) – proces si uchovává čas jako číslo d , které se po každé atomické události inkrementuje. Pokud proces odešle zprávu m , pak do zprávy vloží hodnotu čísla d . Při přijetí zprávy m proces přečte hodnotu času vloženého do zprávy d_{msg} a svoji hodnotu času d upraví následovně: $d = \max(d, d_{msg})$.
- *Vektorový čas* (Vector time) – každý proces uchovává vektor v , který má n položek (n je počet procesů). Při provedení interní události proces p_i inkrementuje hodnotu na pozici v_i , která označuje čas procesu p_i . Pokud proces odesílá zprávu, pak stejně jako u skalárního času přidává do zprávy právě tento svůj čas. Pokud proces přijme zprávu od procesu p_j , pak aktualizuje hodnotu v_j , která značí povědomí procesu p_i o lokálním času procesu p_j .
- *Maticový čas* (Matrix time) – rozšíření vektorového času, kde každý proces uchovává čtvercovou matici řádu n . Matice obsahuje povědomí o vektorovém čase každého procesu.

2.3 Synchronizační nástroje

Při provádění paralelního programu nastávají situace, kdy je nutné procesy nebo vlákna vzájemně řídit (synchronizovat). Pro řízení souběžného přístupu procesů nebo vláken lze použít následující synchronizační prostředky⁷:

- *Semafor* – může být *binární*, nebo *obecný*. Binární semafor obsahuje metody `init(value)`, `lock(value)` a `unlock()`. Proces čeká při zavolání metody `lock()`, dokud hodnota semaforu není 0 a pak jej nastaví na 1. Metoda `unlock()` nastaví hodnotu semaforu na 0 a od blokuje procesy čekající na metodě `lock()` stejného semaforu. Obecný semafor má oproti binárnímu kapacitu, která značí kolik jednotek zdroje chráněného semaforem je k dispozici. Binární semafor je obecný semafor s kapacitou 1. V Javě je obecný semafor implementován třídou `java.util.concurrent.Semaphore` [21, 16].

⁷Pokud není uvedeno jinak, pak lze synchronizační prostředky použít jak pro procesy, tak pro vlákna.

- *Mutex* – binární semafor určený pro vzájemné vyloučení. Při zamknutí mutexu je uložen jeho vlastník a pouze tento vlastník jej může odemknout. V Java SE neexistuje třída reprezentující mutex, nicméně lze pro tento účel upravit třídu `java.util.concurrent.Semaphore` [21, 16].
- *Bariéra* – umožňuje sadě procesů počkat na všechny ostatní. V Javě je implementována třídou `java.util.concurrent.CyclicBarrier` [16].
- *Zámek* – má podobný princip jako semafor, ale jedná se pouze o sdílenou proměnnou a tudíž může řídit přístup pouze mezi vlákny, nikoli procesy [18].
- *Monitor* – abstraktní datový typ, ve kterém jsou sdílené proměnné dostupné pouze přes operace monitoru (včetně jejich inicializace). Tyto operace monitoru jsou vzájemně vyloučené. Monitor lze implementovat v Javě například pomocí tříd `java.util.concurrent.locks.Condition` a `java.util.concurrent.locks.Lock`. Stejně tak lze použít klíčové slovo `synchronized` na libovolný objekt (`java.lang.Object`). Tento objekt obsahuje metody `wait()` (čekání) a `notify()` (Upozornění ostatních, že již nemusí dále čekat.) [21, 16].

Výše uvedené synchronizační prostředky představují pouze základní sadu synchronizačních nástrojů, nicméně další takové nástroje z nich mohou být odvozeny. Synchronizační nástroje slouží k řízení souběžného přístupu, přičemž v další sekci budou ukázány základní chyby, které mohou nastat, pokud není synchronizace provedena správně, nebo dokonce vůbec.

2.4 Chyby v synchronizaci

Chyby v paralelních programech mohou v jednom běhu nastat, zatímco v jiném nemusí, což je dáné proložením běhů jednotlivých vláken nebo procesů. V následujícím textu budou prezentovány některé ze základních chyb tohoto typu včetně příkladů⁸, na kterých bude tento problém demonstrován. Příklady budou vysvětlovány nad vlákny, nicméně pro procesy je situace stejná.

2.4.1 Uváznutí (deadlock)

Prvním chybou, která je zde představena je *uváznutí*. Uváznutí je situace, kdy vlákna čekají na stav, který by mohl nastat, pokud by jedno z těchto vláken mohlo pokračovat. Vlákna jsou takto blokovány navždy [18, 21].

Uváznutí může nastat v příkladu 2.1, pokud 1. vlákno uzamkne zámek A na řádku 6 a ve stejnou chvíli 2. vlákno uzamkne zámek B na tomtéž řádku. V dalším kroku chce 1. vlákno uzamknout zámek B, ale ten je vlastněn 2. vlákнем. Současně chce 2. vlákno uzamknout zámek A, ale ten je vlastněn 1. vlákнем. Vlákna tedy čekají, než se zámky uvolní, což nemůže nikdy nastat. Pokud ovšem nastane takový průběh, že 1. vlákno získá oba zámky (řádky 6 a 7), a až poté chce 2. vlákno uzamknout zámek B (řádek 6), pak k uváznutí nedojde, protože vlákno A dokončí práci a uvolní oba zámky. 2. vlákno pak může oba zámky získat. V tomto příkladu byly popsány 2 průběhy. V jednom k chybě došlo a ve druhém ne. V dalších příkladech už budou demonstrovány jen takové proložení, ve kterých k chybě dojde.

```

1 Lock A~= ...;
2 Lock B = ...;
3
4 // 1. vlakno                                // 2. vlakno

```

⁸Příklady budou psané v pseudokódu vycházejícího z jazyka Java.

```

5 void run() {
6     A.lock();
7     B.lock();
8     B.unlock();
9     A.unlock();
10 }

```

```

void run() {
    B.lock();
    A.lock();
    A.lock();
    B.lock();
}

```

Kód 2.1: Pseudokód příkladu uváznutí dvou vláken.

2.4.2 Blokování

Další chybou je *blokování*, které nastává, pokud vlákno čeká na stav, který generuje jiné vlákno a toto čekání není nutné z hlediska synchronizace. K blokování také dochází, jestliže vlákno čeká na stav, který nemůže nikdy nastat [21].

V příkladu 2.2 může nastat situace, kdy jedno vlákno získá zámek A a už nikdy jej neuvolní. Druhé vlákno tak bude pořád blokováno.

```

1 Lock A~= ...;
2
3 // 1. vlakno
4 void run() {
5     while(true)           // 2. vlakno
6         A.lock();          void run() {
7     }                      while(true)
                           A.lock();
                           }

```

Kód 2.2: Pseudokód příkladu blokování dvou vláken.

2.4.3 Stárnutí

Stárnutí je podobný problém jako blokování, avšak čekání vlákna není shora omezeno. Vlákno čeká na splnění podmínky, která nemusí být nikdy platná v okamžiku testování, ale může nastat situace, kdy platná bude. Pak je vlákno uvolněno a může pokračovat. Pokud by uvolnění nemohlo nikdy nastat, tak by se jednalo o blokování [21].

2.4.4 Časově závislá chyba nad daty

K časově závislé chybě nad daty může dojít, pokud dochází k souběžnému přístupu více vláken ke sdílené proměnné alespoň jeden z přístupů je zápis [21].

V příkladu 2.3 se nachází sdílená proměnná *value*, která reprezentuje aktuální hodnotu. Dále se zde nachází dvě vlákna, kde 1. vlákno zapisuje aktuální hodnoty a 2. vlákno vypisuje aktuální hodnoty na výstup. Pokud jsou obě vlákna spuštěny současně, pak může dojít ke dvěma výsledků. Na výstupu se objeví hodnota 5 nebo 8 podle toho, zda bude nejdříve aktuální hodnota vypsána, nebo aktualizována.

```

1 int value = 5;
2
3 // 1. vlakno
4 void run() {
5     value = 8;           // 2. vlakno
6 }                         void run() {
                           print(value);
                           }

```

Kód 2.3: Pseudokód příkladu časově závislé chyby nad daty.

2.4.5 Porušení atomicity

K porušení atomicity dojde, pokud vlákno získá stav nějaké sdílené proměnné, další výpočet vlákna závisí na tomto stavu a zároveň během tohoto výpočtu k této proměnné může přistoupit jiné vlákno a její stav změnit [21].

Následující příklad vychází z předchozího příkladu. K porušení atomicity dojde, pokud 2. vlákno přečeť stav proměnné `value` a uloží si jej do proměnné `tmp`. Následně 1. vlákno aktualizuje hodnotu `value` na hodnotu 8 a nakonec 1. vlákno provede také aktualizaci této proměnné, ale již na základě neplatného stavu uloženého v proměnné `tmp`. Na výstupu se tak může objevit některá z množiny hodnot: 6, 8, 9.

```
1 volatile int value = 5;
2
3 // 1. vlakno                                // 2. vlakno
4 void run() {                               void run() {
5     value = 8;                           int tmp = value;
6 }                                         value = tmp + 1;
7                                         print(value);
8 }
```

Kód 2.4: Pseudokód porušení atomicity.

2.4.6 Porušení kontraktu

Poslední zde uvedenou chybou je *porušení kontraktu* [15]. Kontrakt je sekvence veřejných metod objektu, která musí být vykonána atomicky, s ohledem na ostatní veřejné metody stejného objektu. Problematika kontraktů je dále popsána v kapitole 3.2.1.

2.5 Metody testování vícevláknových programů

V této kapitole jsou popsány metody testování vícevláknových aplikací. Největší důraz je zde kladen na *dynamickou analýzu* a metodu *vkládání šumu*, neboť tyto dvě metody souvisí s dalšími částmi této práce.

2.5.1 Statická analýza

Statická analýza zkoumá software bez jeho spouštění. Tato metoda je používána například pro hledání syntaktických chyb a je typicky spouštěna před překladem. Nicméně i tuto metodu je možné využít pro testování paralelních programů. Nevýhodou této analýzy je produkování velkého množství *false alarmů*⁹, neboť statická analýza nemá k dispozici informace o konkrétních instancích objektů. Existují ovšem metody založené na provádění testovaného softwaru, které se tento problém snaží odstranit. Jednou z nich je metoda *dynamické analýzy* [6].

2.5.2 Dynamická analýza

Dynamická analýza zkoumá software na základě jeho provádění, během kterého shromažďuje informace o jeho běhu (události a jejich uspořádání, stavy zámků, vláken, paměťových míst, atd.). Kromě hlášení chyb, které v daném běhu nastaly se dynamická analýza snaží extrapolovat nasbírané informace a odhalit chyby, které nenastaly, ale v jiném proložení nastat mohou. Přesto dynamická

⁹*False alarm* je varování, že se může vyskytnout chyba, přestože ve skutečnosti nikdy vzniknout nemůže.

analýza nedokáže odhalit všechny chyby, ale pouze ty, které lze odvodit z běhů, kterých byla svědkem. Z tohoto důvodu bývá tato analýza spouštěna opakovaně a kombinována například s metodou *stochastického vkládání šumu* nebo *deterministického testování*. Díky témtu metodám je zvyšován počet testovaných proložení programu, a tím i pravděpodobnost nalezení chyb. Samotná analýza zatěžuje systém, což má podobný efekt jako vkládání šumu. Tento jev se označuje jako *noise effect* a je třeba s ním při analýze počítat. Dynamická analýza je většinou zaměřena pouze na určitý typ chyb a v takovém případě sbírá pouze informace související s tímto typem. U různých typů dynamické analýzy se rozlišují dvě vlastnosti *sound* a *precise*. Pokud dynamická analýza splňuje první vlastnost, pak nemůže přehlédnout chybu. Pokud splňuje druhou vlastnost, pak neprodukuje false alarmy. Nicméně platí, že dynamická analýza nemusí splňovat ani jednu z nich, tj. může přehlížet chyby a zároveň může produkovat false alarmy. Programy provádějící dynamickou analýzu se nazývají *dynamické analyzátor* (viz kapitola 3) a jejich příkladem jsou: *Eraser*, *GoldiLocks*, *FastTrack* nebo *DJIT+* [6, 20, 8, 9].

2.5.3 Deterministické testování

Deterministické testování je metoda založená na opakovaném provádění testovaného softwaru, přičemž má plnou kontrolu nad jeho prováděním. K tomuto účelu je použitý *deterministický plánovač*, který je implementován například pomocí vkládání silného šumu. Cílem je otestovat co nejvíce možných proložení vláken. V každém kroku provádění softwaru analyzátor zkoumá, jaké možnosti v plánování mohou nastat a ty ukládá do stavového prostoru testovacích scénářů. Při dalších běžích se provádí další scénáře z tohoto stavového prostoru, které se opět dále větví. Z tohoto důvodu jsou zavedeny některé heuristiky, které omezují velikost stavového prostoru. Stavový prostor všech možných scénářů je příliš velký a u větších programů by testování trvalo příliš dlouho. Často se tak využívá například testování, kdy plánovač nechává běžet vždy pouze jedno vlákno a ostatní nechává pozastavené, nebo je například omezen maximální počet přepnutí kontextu. Případ, kde se testují všechny proložení, se nazývá *full model checking* [6].

2.5.4 Zátěžové testování

Zátěžové testování je založeno na principu vytvoření nejhoršího možného prostředí, ve kterém aplikace může běžet. V souvislosti s hledáním konkurenčních chyb je to vytvoření velkého množství vláken, které budou navzájem soupeřit o sdílené zdroje. Tímto přístupem mohou být odhaleny některé chyby, ale s největší pravděpodobností se bude jednat o chyby častěji se vyskytující. Vzhledem k tomu, že proložení prováděné testovaným softwarem jsou náhodné, může toto testování způsobovat opakované provádění již prozkoumaných proložení, přestože existují jiné proložení, které otestovány nebyly. Další nevýhodou je značné zatížení jak testovaného softwaru, tak prostředí, kde testování běží [6].

2.5.5 Vkládání šumu

Tento typ testování vkládá šumu (*noise*) [12] do prováděného kódu, který opožďuje vlákna, a tak může dojít k proložení, které by jinak nastalo pouze velmi výjimečně. Stejně tak mohou být odhaleny chyby souběžného přístupu dvou instrukcí, které jsou v kódu dostatečně daleko od sebe a za normálních okolností by nebylo prakticky možné, aby se vykonaly souběžně. Tato chyba může být odhalena vložením dostatečně silného¹⁰ šumu do jednoho z vláken. Pomocí metody vkládání šumu může být prozkoumáno velké množství scénářů v relativně krátké době. Náhodné vkládání šumu

¹⁰Silným šumem pozastaví provádění vlákna na delší dobu.

nemusí být příliš efektivní, protože může docházet k vkládání na místa, které nijak neovlivňují proložení vláken a šum by tak pouze zatěžoval systém. Lepších výsledků je dosaženo v případě použití některé heuristiky, která určuje vkládání šumu pouze na specifická místa v kódu (*noise seeding problem*) [6]. Stejně důležitým faktorem, jako je umístění šumu, je i vkládání vhodného typu šumu (*noise seeding problem*). Typ je dán sílou šumu a operací, která šum představuje [6].

V programovacím jazyce Java lze použít například funkce [16]:

- `yield()` – způsobí přepnutí kontextu (síla udává počet zavolání této funkce, než může vlákno pokračovat),
- `sleep()` – blokuje (uspí) vlákno po zadanou dobu (síla šumu),
- `wait()` – jako `sleep()` s rozdílem, že vlákno čeká na objektu *monitoru*.

2.6 Nástroje pro testování Java programů

Pro testování paralelních programů v jazyce Java existuje celá řada programů, a proto zde budou představeny především ty, jež souvisejí s dynamickou analýzou, která je předmětem této práce. Jedním z nejznámějších je *Java Pathfinder* (JPF) [2], který byl vyvinut v *NASA Ames Research Center*. JPF dokáže vykonávat testovaný program a ukládat, porovnávat a obnovovat stavy tohoto programu. Díky těmto vlastnostem je používán jako tzv. *model checker*, ale lze v něm také na definovat dynamické analyzátoru. Podobné nástroje jako JPF jsou například *Bandera* nebo *CBMC*, který byl původně vyvinut pro C/C++ a rozšířen o podporu jazyka Java.

Dalším nástrojem je projekt *IBM ConTest* [3] sloužící pro instrumentaci a dynamickou analýzu Java programů. Tento nástroj je velmi podobný dalšímu nástroji *RoadRunner* [9] (viz kapitola 2.6.1), který lze rovněž využít pro instrumentaci a dynamickou analýzu. V neposlední řadě existuje nástroj *Java Race Detector & Healer*¹¹ vyvinutý výzkumnou skupinou VeriFIT, který slouží pro detekci časově závislých chyb. Tento projekt využívá pro instrumentaci programu zmíněný IBM ConTest.

Jak bylo vysvětleno v kapitole 2.5.2, tak dynamická analýza nedokáže odhalit všechny chyby během jednoho běhu, a proto je nutné ji spouštět opakováně. Pro tento účel slouží nástroj *SearchBestie* [11], který byl opět vyvinut výzkumnou skupinou VeriFIT. SearchBestie je společně s nástrojem RoadRunner využívána v další části této práce, a proto budou nyní popsány detailněji.

2.6.1 RoadRunner

RoadRunner [9] je framework navržený pro dynamickou analýzu vícevláknových Java programů, který je také napsán v jazyce Java. *RoadRunner* vkládá instrumentační kód do bytekódu testovaného programu, což umožňuje testovat programy i bez znalosti zdrojových kódů. Vložený instrumentační kód generuje *tok událostí*, které nastávají v testovaném programu. Těmito událostmi jsou například:

- události synchronizace na zámcích,
- přístup k proměnným,
- vytvoření/ukončení vláken,
- vstup/výstup do/z metod a další.

¹¹Nástroj dostupný na adrese <http://www.fit.vutbr.cz/research/groups/verifit/tools/racedetect/>

Tok událostí je zpracováván pomocí *nástrojů*, které mohou filtrovat události a je možné je skládat do tzv. *řetězce nástrojů*. Tímto způsobem lze skládat složitější dynamické analyzátorы z jednodušších kroků, kde každý krok je reprezentován jednodušším nástrojem.

Nástroje jsou stavební kameny všech analyzátorů implementovaných v tomto frameworku. Při jejich vytváření je nutné rozšířit třídu obecného nástroje, tedy třídu Tool. Tato třída definuje metody pro zpracování všech událostí (tzv. *handlery událostí*) z toku událostí. Každá tato metoda má jako parametr objekt reprezentující zachycenou událost. Pokud určitý nástroj nezpracovává události určitého typu, pak tyto události musí přeposlat dalšímu nástroji v řetězci nástrojů. Tabulka ?? obsahuje vybrané nástroje, které již jsou v tomto frameworku implementovány.

Název nástroje	Popis
ThreadLocal	Filtuje přístup k lokálním datům.
ReadOnly	Filtuje přístup k datům určeným pouze pro čtení.
ProtectingLock	Filtuje operace nad zámky, které jsou chráněny dalšími zámky.
LockSet	Detekuje časově závislé chyby nad daty s použitím LockSet algoritmu.
EraserWithBarrier	Detekuje časově závislé chyby nad daty s použitím LockSet algoritmu a analýzy bariér.
HappensBefore	Detekuje časově závislé chyby nad daty s použitím algoritmu VectorClock.
DJIT+	Detekuje časově závislé chyby nad daty s použitím optimalizovaného algoritmu VectorClock.
MultiRace	Detekuje časově závislé chyby nad daty s použitím hybridní LockSet/VectorClock analýzy.
Goldilocks	Detekuje časově závislé chyby nad daty s použitím rozšířeného LockSet algoritmu.
FastTrack	Detekuje časově závislé chyby nad daty s použitím FastTrack algoritmu.

Tabulka 2.1: Tabulka obsahující vybrané implementované nástroje v projektu RoadRunner [11]

Pro každý objekt *Thread*, který využívá JVM pro reprezentaci vláken, vytváří *Roadrunner* objekty typu *ShadowThread*, které jím přísluší¹². Stejným způsobem vytváří také objekty *ShadowLock* pro každý objekt použitý jako zámek a objekt typu *ShadowVar* pro každou lokaci v paměti. Poslední jmenovaný objekt je uložen v tzv. *shadow location*, která koresponduje s každou lokací v paměti. Zmíněné objekty pak mohou být využity pro uložení specifických informací důležitých pro konkrétní nástroj. Každý nástroj implementovaný v *RoadRunneru* by měl splňovat následující podmínky [9]:

1. Každý handler události musí vyvolat stejný handler v dalším nástroji v řetězci.
2. Pro identifikaci, který nástroj vlastní *shadow location*, musí nástroj obsahovat *shadow objekt* typu *T* a dále může ukládat do *shadow location* pouze objekty tohoto typu *T*.
3. Metoda *makeShadowVar*¹³ každého nástroje musí vracet objekt typu *T*.
4. Pokud je vyvolán *access handler*, tj. handler přístupu do paměti a *shadow location* je vlastněna tímto nástrojem, tj. *shadow location* obsahuje objekt typu *T*, pak nástroj musí provést jednu z následujících možností:

¹²Objekt *ShadowThread* obsahuje referenci na originální objekt *Thread*

¹³Metoda *makeShadowVar* je zavolána při prvním přístupu do paměti na danou lokaci. Tj. při prvním přístupu k proměnné.

- Udržet vlastnictví paměťového místa tím, že uloží/ponechá objekt typu T v *shadow location*.
 - Vzdá se vlastnictví paměťového místa pomocí metody *advance*, která nahradí *shadow location* objektem typu T_1 , kde T_1 značí typ *shadow* objektu následujícího nástroje. *Shadow location* už nikdy nesmí obsahovat objekt typu T .
5. Jestliže je vyvolán *access* handler a paměťové místo není vlastněno tímto nástrojem, pak musí nástroj zavolat *access* handler následujícího nástroje.

Body 4. a 5. říkají, že *shadow location* každého paměťového místa může modifikovat pouze nástroj, který je jeho aktuálním vlastníkem.

2.6.2 SearchBestie

Platforma *SearchBestie* [11] je platforma určená pro hledání optimálních parametrů testů a jejich spouštění napsaná v jazyce Java. Využívá techniky prohledávání stavového prostoru, jež tvoří kombinace parametrů, se kterými jsou jednotlivé testy spouštěny. Lze ji mimo jiné použít pro nalezení optimálních testů při testování vícevláknových aplikací, čehož bude využito v této práci. *SearchBestie* nejprve využívala pro instrumentaci programů nástroj IBMContest, který byl později nahrazen nástrojem RoadRunner (viz kapitola 2.6.1). Jejím úkolem je hledat optimální parametry a s nimi spouštět testování programu právě prostřednictvím nástroje RoadRunner. *SearchBestie* se skládá z následujících modulů:

- *Manager* – řídí celý proces a nabízí pomocné funkcionality.
- *Search* – modul vybírá kombinaci parametrů a test, který má být v příštím kroku vykonán.
- *Executor* – vykonává testy s vybranými parametry a sbírá výsledky.
- *Storage* – ukládá výsledky testů.
- *Analysis* – analyzuje výsledky uložené modulem *Storage*.

SearchBestie na vstupu přijímá konfigurační soubor, ve kterém jsou definovány mimo jiné parametry ovlivňující vkládání šumu. Kombinace těchto parametrů vytváří stavový prostor možných konfigurací testů, který *SearchBestie* postupně prochází. Tyto parametry jsou:

- *NoiseFrequency* – udávající sílu šumu.
- *NoiseStrength* – udávající frekvenci šumu.
- *NoiseType* – udávající typ vkládaného šumu.

Na základě jednotlivých konfigurací (tj. stavů) ve stavovém prostoru vkládá RoadRunner šum do testovaného programu, a tím ovlivňuje možné proložení vláken.

Ve vstupním souboru lze také definovat další parametry, mezi které patří:

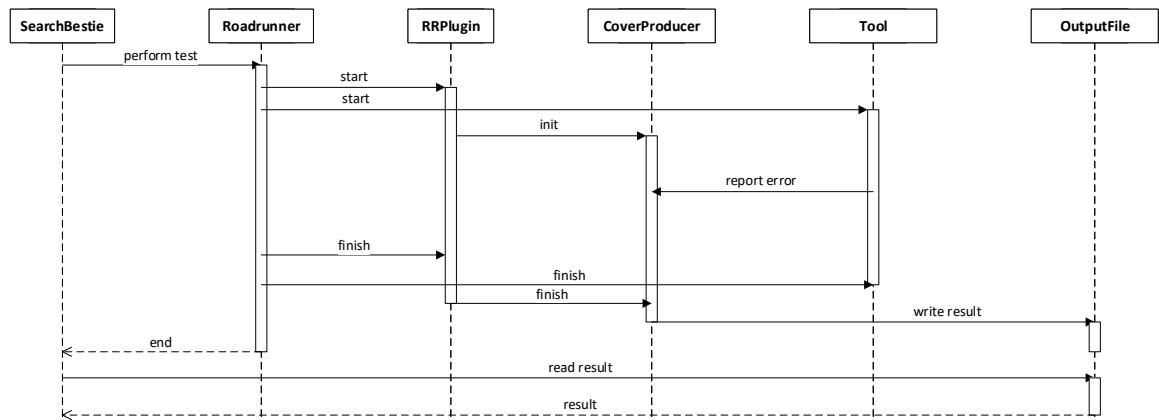
- *Searcheng* – definuje, jakým způsobem se bude prohledávat stavový prostor.
- *Executor* – v tomto elementu se nachází nastavení pro modul *Executor*. Je zde možné například nastavit maximální počet spuštění testů.

- *Fitness* – definuje výpočet fitness funkce.
- *Javatest* – tento element definuje, jaký typ testu bude spuštěn a nastavení tohoto testu. Vzhledem k použití RoadRunneru se jedná o *RoadRunnerTest* a nastavení všech parametrů, které jsou potřebné pro jeho spuštění.

2.6.3 Propojení SearchBestie a frameworku Roadrunner

SearchBestie je nástroj určený pro hledání optimálních parametrů testů a jejich spuštění. Samotné testy jsou prováděny samostatným nástrojem, kterým je v tomto případě RoadRunner. SearchBestie následně sbírá výsledky z těchto testů a na jejich základě vytváří stavový prostor. Pro propojení je důležitá zejména třída *RRPlugin* v RoadRunneru, která musí být použita jako první nástroj v řetězci nástrojů. Tato třída obsahuje tzv. *Cover proucery* (dále jen CP), které zaznamenávají informace o chybách vyskytujících se v testovaném programu. Při ukončení testu všechny CP zapíší všechny nasbírané informace o chybách do výstupních souborů, které jsou následně načteny v Searchbestie, a jejich obsah je dále zpracován. Každý takový CP může zaznamenávat různé informace a ty musejí být zpracovány různým způsobem. Z toho důvodu musí existovat odpovídající tzv. *Result item creator* (dále jen RIC), který dokáže tyto výsledky zpracovat.

V diagramu 2.1 je znázorněna zjednodušená vzájemná komunikace mezi těmito komponentami. SearchBestie nejprve spustí RoadRunner s patřičnými parametry pomocí metody *performTest*. Následně RoadRunner spustí RRPlugin a další nástroje (reprezentované v diagramu objektem *Tool*), zadané v parametrech. RRPlugin pro každý další nástroj vytvoří patřičný CP. Jakmile dojde k chybě, která byla detekována některým nástrojem, pak tento nástroj reportuje chybu odpovídajícímu CP. Na konci testování jsou ukončovány jednotlivé nástroje a v této chvíli dá RRPlugin pokyn všem CoverProducerům, aby zapsaly všechny nasbírané informace do výstupního souboru. Jakmile SearchBestie dostane informaci o ukončení RoadRunneru, načte a zpracuje informace z výstupních souborů jednotlivých CP.



Obrázek 2.1: Komunikace v systému.

Kapitola 3

Dynamické analyzátory

Dynamické analyzátory jsou programy provádějící dynamickou analýzu, přičemž každý z nich je zaměřen na hledání určitého typu chyb. Typickým problémem je hledání časově závislých chyb nad daty. Analyzátorů, které hledají tento typ chyb, je hned celá řada. Nejjednodušším z nich je algoritmus *Eraser*, který zkoumá program s pomocí množin zámků držených jednotlivými vlákny, přičemž princip tohoto algoritmu dále rozšiřuje další analyzátor *Goldilocks*. Dalšími analyzátorami jsou *DJIT+* nebo *FastTrack*, které hledají chyby za pomoci vektorového času. Příkladem dynamického analyzátoru, který hledá jiný typ chyb, může být analyzátor kontraktů. Právě tento analyzátor společně s analyzátorem *FastTrack* je dále více popsán [8, 20, 7].

3.1 Analyzátor FastTrack

FastTrack [8] analyzátor patří do skupiny *precizních*¹ analyzátorů a je určen pro odhalování časově závislých chyb nad daty. Využívá adaptivní reprezentaci relace Happens before, díky které dokáže významně zvýšit svou rychlosť oproti jiným precizním analyzátorům, jako jsou BasicVC nebo DJIT+. Běžné precizní dynamické analyzátoru ukládají pro každé vlákno vektorový čas po sledního zápisu do každé proměnné x . Vektorový čas je n -tice čísel, kde n je počet vláken (viz kapitola 2.2). Všechny operace nad vektorovým časem mají tudíž složitost $O(n)$. Oproti tomu FastTrack může ukládat pouze informaci o posledním zápisu do každé proměnné x napříč všemi vlákny, a to v případě, že všechny zápisy do proměnné x jsou uspořádané relací Happens before. Tato informace, nazývaná *epocha*, se skládá z času posledního zápisu a identifikátoru vlákna, ve kterém zápis proběhl. Veškeré operace nad touto epochou je následně možné provést v konstantním čase $O(1)$. Stejně jako pro zápis ukládá FastTrack informaci pouze o posledním čtení každé proměnné x , pokud jsou opět všechny předchozí čtení uspořádány pomocí relace Happens before. Pokud nastane případ, kdy některé čtení nebo zápisy nejsou touto relací uspořádány, FastTrack ukládá informaci o časech vykonání těchto operací pomocí vektorových hodin [8].

Jelikož nutnost ukládat plný vektorový čas je pouze v minimálním množství situací, dokáže FastTrack díky použití epoch snížit režii spojenou s analýzou z $O(n)$ až na $O(1)$. Jak bylo popsáno výše, FastTrack analyzátor ukládá informace o posledním zápisu pomocí epochy. Vlákna ovšem pro reprezentaci logického času používají vektorový čas, a proto je nutné tyto dvě formy umět porovnat. *Epocha* je definována jako dvojice $c@t$, kde c je logický čas vlákna t , a platí, že tato epocha předchází vektorový čas V (označováno jako $c@t \leq V$) v relaci Happens before, jestliže platí $c \leq V(t)$ [8].

¹Dynamický analyzátor je precizní, pokud splňuje vlastnost *precise* (viz kapitola 2.5.2)

Analyzátor rozlišuje následující 3 typy situací, které mohou nastat při přístupu ke sdílené proměnné [8]:

- čtení s následným zápisem,
- zápis s následným čtením,
- zápis následující dalším zápisem.

3.1.1 Zápis následovaný čtením nebo zápisem

Detekování dvou konkurentních zápisů je pomocí porovnání epochy s vektorovým časem jednoduché. Označíme-li epochu posledního zápisu do proměnné x jako W_x , pak ve chvíli následujícího zápisu do této proměnné v čase V stačí porovnat tento vektorový čas s epochou W_x . Stejným způsobem lze také rozhodnout, zda mohlo dojít k časově závislé chybě nad daty při zápisu následovaném čtením. Ve chvíli čtení proměnné x v čase V , se opět porovná tento vektorový čas s epochou posledního zápisu W_x , tedy zda platí: $W_x \leq V$. Pokud tato rovnost platí, nedošlo k časově závislé chybě nad daty [8].

3.1.2 Čtení následované zápisem

Detekování možného souběžného přístupu při čtení s následným zápisem není v analyzátoru Fast-Track tak jednoduché, jako předchozí dva případy. Důvodem je to, že jednotlivá čtení libovolné proměnné x nemusejí být totálně uspořádána, jako tomu je u zápisu². Nastane-li případ, že některá čtení nejsou totálně uspořádána, pak je nutné uchovat celý vektorový čas tohoto čtení. Tento případ může nastat pouze v případě, kdy je proměnná x sdílena a kdy čtení není chráněno zámkem. V ostatních případech postačí uchování epochy. To, zda došlo k souběžnému přístupu, lze opět určit podobně jako v předchozích případech, ale může nastat stav, kdy budou porovnávány dva vektorové časy, konkrétně R_x (čas posledního čtení proměnné x) a aktuální čas zápisu V . V tomto případě bude porovnání časově nejnáročnější, nicméně k této situaci dochází pouze velmi zřídka [8].

3.2 Analyzátor kontraktů

Tato kapitola nejprve popisuje samotné kontrakty a jejich rozšíření. Následně je představena dynamická analýza, která umožňuje jejich detekci.

3.2.1 Definice základního kontraktu

Kontrakt [15] byl původně definován jako sekvence příkazů s definovanými podmínkami. Je-li sekvence vykonána bez splnění těchto podmínek, dochází k porušení tohoto kontraktu. *Kontrakt pro souběžnost* [4] je proti tomu protokol přístupu veřejných služeb modulu (tj. veřejných metod). Každý modul může mít definován vlastní kontrakt, který obsahuje množinu sekvencí služeb (metod). Podmínkou splnění kontraktu je atomické vykonání těchto sekvencí, pokud jsou vykonávány nad stejným objektem.

Pro další popis je nutné zavést následující značení, které vychází z článků [7] a [5]. Nechť je množina jmen veřejných metod modulu označena jako Σ_M , dále kontrakt jako množina klauzulí R , kde každá klauzule $\varrho \in R$ je regulární výraz nad Σ_M . K porušení kontraktu dojde, není-li sekvence

²Pokud dojde k porušení totálního uspořádání u zápisu, pak byla nalezena časově závislá chyba nad daty.

reprezentovaná kontraktem provedena atomicky, tj. sekvence konaktu je proložena alespoň jednou metodou z množiny Σ_M ³.

Příkladem konaktu (označme jej ϱ_1) může být sekvence metod `indexOf` a `get` nad objektem typu *seznam*. Jestliže by tato sekvence byla proložena například metodou `add`, která by vložila prvek na první místo seznamu, pak by index hledaného prvku vrácený metodou `indexOf` již nebyl platný a následné použití metody `get` by vrátilo jiný prvek seznamu. Tento konakt lze zapsat jako:

$$\varrho_1 : \text{indexOf} \quad \text{get}. \quad (3.1)$$

3.2.2 Rozšíření konaktu

Rozšíření publikované ve článku [5] vychází z předchozí definice konaktu, přičemž spojuje kontrobu atomicity sekvencí konaktu s definicí metod, které tuto sekvenci nesmí porušit. Znamená to tedy, že sekvence metod musí být atomická pouze s ohledem na proložení pouze určenou množinu metod. Dalším rozšířením je přidání parametrů do jednotlivých metod.

Uvažujme nejprve *rozšíření konaktu o parametry*. Zaměříme-li se znovu na příklad konaktu ϱ_1 , pak je viditelné, že porušení konaktu způsobí chybu pouze v případě, kdy je návratová hodnota metody `indexOf` použita jako parametr metody `get`. Tento konakt s rozšířením o parametry lze zapsat jako:

$$\varrho'_1 : X = \text{indexOf}(_) \quad \text{get}(X). \quad (3.2)$$

Dále uvažujme *rozšíření o kontextové informace* (tj. rozšíření o metody, které nesmí sekvenci konaktu porušit). Uvažujme opět příklad konaktu ϱ_1 a základní definici konaktu. Tato definice říká, že sekvence konaktu nesmí být proložena žádnou metodou z množiny veřejných metod modulu Σ_M . Uvažujme tedy například metody `indexOf` a `remove` z množiny Σ_M . Bude-li sekvence konaktu ϱ_1 proložena voláním metody `indexOf`, pak k žádné chybě nedojde. Bude-li ovšem proložena voláním metody `remove`, pak k chybě může dojít. Toto rozšíření umožňuje definovat množinu sekvencí metod (sekvence těchto metod bude dále označována jako *spoiler*), vůči kterým musí být sekvence konaktu (dále označovaná jako *target*) provedena atomicky [5].

Nechť je tedy \mathbb{R} množina všech *targetů*, kde každý target $\varrho \in \mathbb{R}$ je regulární výraz nad množinou Σ_M . Nechť \mathbb{S} je množina *spoilerů*, kde každý spoiler $\sigma \in \mathbb{S}$ je regulární výraz nad Σ_M . Dále označme abecedu všech targetů jako $\Sigma_R \subseteq \Sigma_M$ a všech spoilerů jako $\Sigma_S \subseteq \Sigma_M$. Pak *konakt* je definován jako relace $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$, kde pro každý target je definována množina spoilerů, které mohou vyvolat porušení atomicity.

K porušení konaktu dojde, je-li nějaká sekvence prováděných metod r , odpovídající targetu $\varrho \in \mathbb{R}$ a vykonána nad objektem o , plně proložena celou sekvencí metod s , odpovídající nějakému spoileru $\sigma \in \mathbb{C}(\varrho)$, která je vykonána nad stejným objektem o . Sekvence targetu r je plně proložena sekvencí spoileru s , jestliže začátek vykonávání r začíná před začátkem vykonávání s a současně konec vykonávání s předchází konec vykonávání r . [5]

V další části této práce budou uvažovány obě tyto rozšíření, přičemž značení konaktu bude například pro výše zmíňovaný příklad následující:

$$\varrho''_1 : X = \text{indexOf}(_) \quad \text{get}(X) \rightsquigarrow \text{remove}(_). \quad (3.3)$$

Konakty lze detektovat jak na základě statické analýzy, tak na základě dynamické analýzy. Vzhledem k tématu této práce zde bude popsána pouze dynamická analýza, která byla navržena jak pomocí LockSet algoritmu, tak pomocí relace Happens before. Tato dynamická analýza byla také implementována v projektu ANaConDA, která zkoumá paralelní programy psané v jazyce C/C++.

³Volání všech metod se předpokládá nad stejným objektem o

3.2.3 Dynamická analýza založena na LockSet algoritmu

Prvním možným způsobem detekce kontraktů pomocí dynamické analýzy je analýza založená na množině zámků držených jednotlivými vlákny (*LockSet-based*), jejíž typickým představitelem je algoritmus Eraser. Díky tomuto typu dynamické analýzy je možné rozšířit proložení, které se událo, a tím nalézt porušení kontraktu, které se přímo nestalo. Tento způsob byl představen ve článku [7] a dokáže pracovat pouze ze základní definicí kontraktu.

V této analýze je nejdříve nutné detektovat kontrakty, které se vyskytují ve vykonávaném programu. Tohoto je dosaženo pomocí konečných automatů, kde pro každou sekvenci kontraktu je vytvořen odpovídající konečný automat. Každé vlákno obsahuje množinu instancí těchto automatů, kde instance odpovídají aktuálním nedokončeným sekvencím kontraktu, které se nachází v aktuálním stavu programu. Je-li zavolána metoda $m \in Q_M$, pak je nad každou instancí automatu proveden pokus o postup do dalšího stavu pomocí této metody m (tentotok je označován jako *advance*). Jestliže je nový stav automatu stavem konečným, pak je detektován výskyt sekvence kontraktu (tj. výskyt targetu, nebo spoileru). Je-li metoda m metodou, kterou začíná některá sekvence metod kontraktu, pak je vytvořena nová instance kontraktu [7].

Je-li sekvence kontraktu detektována, je nutné ověřit, zda byla prováděna atomicky. Pomocí informací o zámcích, držených jednotlivými vlákny během provádění kontraktu, lze detektovat, zda byla celá sekvence chráněna alespoň jedním zámkem. Jestliže byla chráněna alespoň jedním zámkem, pak je pravděpodobné, že byla provedena atomicky. Pravděpodobně to je proto, že může nastat případ, kdy tomu tak nebude. Konkrétně se jedná o situaci, kdy dojde k proložení dvou sekvencí kontraktu nad stejným objektem ve dvou různých vláknech a obě tyto sekvence budou chráněny zámkem, ale tento zámek bude odlišný. Pokud nastane tento případ, mohou být sekvence prováděny souběžně, a tím dojde k porušení kontraktu. Tato situace je diskutována ve článku [7] a jako řešení je navržena dynamická analýza založena na relaci Happens before.

3.2.4 Dynamická analýza založena na relaci Happens before

Dynamická analýza založena na relaci Happens before, představena ve článku [5], je navržena pro rozšířenou definici kontraktu s kontextovými informacemi. Tato analýza využívá Happens before relaci (viz kapitola 2.1), kde jako komunikace mezi vlákny jsou uvažovány operace *acquire* a *release* nad stejnými zámkami a operace *fork* a *join*. Tato analýza umožňuje detekci porušení kontraktů za běhu díky technice nazvané *trace window*. Tento koncept umožňuje neuchovávat celou sekvenci všech provedených metod, nýbrž pouze podmnožiny této sekvence (dále označováno jako *okno běhu*), která se postupně pohybuje v průběhu vykonávání programu. Cílem tohoto konceptu je uchovávat okno o co nejmenším počtu naposled vykonaných metod. Události se přidávají do okna běhu v v momentě, kdy se vyskytnou, a odebírají, jestliže již nejsou potřeba. Odebírání je definováno tak, že všechny události z určité instance targetu (spoileru) se mohou odebrat, pokud již nepatří do žádné jiné aktuálně sledované instance targetu, nebo spoileru. Velikost okna tak závisí pouze na počtu instancí targetů a spoilerů, nikoli na délce běhu programu. Aby bylo možné odstraňovat z tohoto okna události, je nutné odstraňovat nepotřebné instance targetů a spoilerů. Pro odstraňování těchto instancí jsou definovány následující pravidla [5]:

- Odstranění instance spoileru s je bezpečné, jestliže porušení kontraktu, které může být odhaleno pomocí instance s , je odhalitelné také bez této instance s .
- Pro každé vlákno a každý spoiler je nutné uchovávat pouze poslední instanci spoileru.
- Existují-li dvě instance r_1, r_2 targetu $\varrho \in \mathbb{R}$ takové, že $end(r_1) \rightarrow_{hb} start(r_2)$, pak je bezpečné odstranit instanci r_1 , jestliže s začíná až za *oknem běhu* nebo jestliže $start(s) \rightarrow_{hb} start(r_1)$.

- Existují-li dvě instance r_1, r_2 targetu $\varrho \in \mathbb{R}$ takové, že $end(r_1) \rightarrow_{hb} start(r_2)$. A necht' s je instance spoileru $\sigma \in \mathbb{S}$, kde $(\varrho, \sigma) \in \mathbb{C}$, pak je bezpečné odstranit instanci r_1 s ohledem na s , pokud platí, že $start(s) \in v \wedge end(s) \notin v$ a zároveň $start(s) \not\rightarrow_{hb} start(r_2)$.
- Obecně platí, že pro každý target $\varrho \in \mathbb{R}$ je uchováno $|T| + 1$ instancí, kde T značí množinu běžících vláken v *okně běhu*. Pro každé vlákno musí být uchována jedna instance a také 1 instance navíc pro vlákno, které může být potencionálně vytvořeno.

Kapitola 4

Návrh analyzátoru kontraktů

V této kapitole je popsán návrh dynamického analyzátoru kontraků, který odpovídá popisu v kapitole 3.2. Jedná se o analýzu kontraktů s parametry, přičemž pro detekci většího počtu chyb byla využita relace Happens before. Analyzátor kontraktů je navržen jako nástroj ve frameworku RoadRunner, přičemž jej lze spustit jak samostatně, tak z platformy SearchBestie, která je schopna jej spustit a zpracovat jeho výsledky.

4.1 Specifikace požadavků

Specifikace požadavků obsahuje funkční požadavky na analyzátor kontraktů, ale také požadavky související s propojením tohoto analyzátoru a platformy SearchBestie.

1. Obecné požadavky
 - 1.1. Analyzátor musí být implementován v projektu RoadRunner.
 - 1.2. Roadrunner s analyzátem musí být spustitelný z příkazové řádky.
 - 1.3. Roadrunner s analyzátem musí být spustitelný ze SearchBestie.
 - 1.4. RoadRunneru musí vytisknout informace o nalezených chybách na chybový výstup.
 - 1.5. RoadRunneru musí být schopný uložit informace o nalezených chybách do výstupního souboru.
 - 1.6. SearchBestie musí zpracovat tyto výstupní informace.
 - 1.7. SearchBestie musí na základě nalezených chyb rozšiřovat stavový prostor parametrů testů a vybírat vhodné parametry pro další spuštění.
2. Funkční požadavky
 - 2.1. Analyzátor získá definici kontraktů z konfiguračního souboru.
 - 2.2. Musí být definován formát konfiguračního souboru.
 - 2.3. Cesta ke konfiguračnímu souboru musí být předána analyzátoru jako parametr v příkazové řádce.
 - 2.4. Cesta ke konfiguračnímu souboru musí být předána SearchBestie ve vstupním souboru, která ji předá RoadRunneru.
 - 2.5. Analyzátor musí být schopný zpracovat konfigurační soubor a uložit si kontrakt do vnitřní reprezentace.

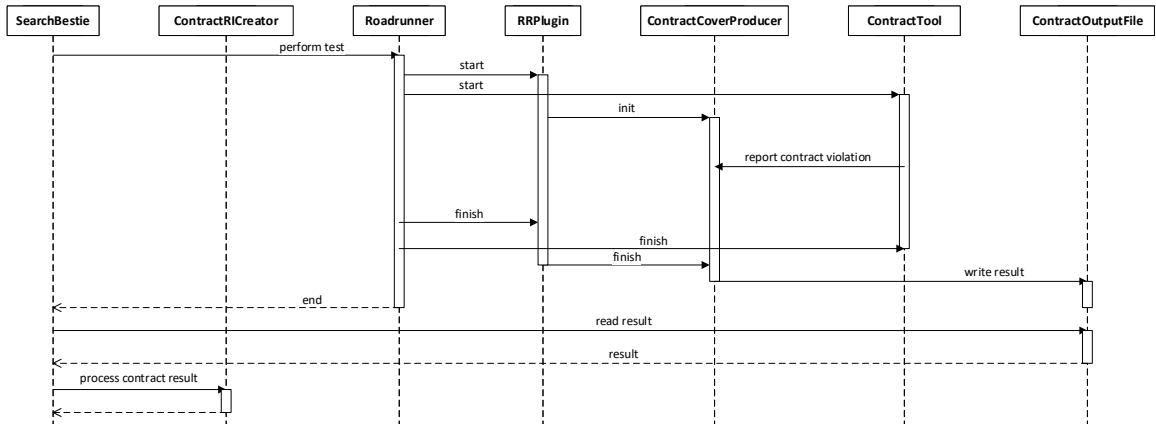
- 2.6. Porušení kontraktu musí být detekováno v následujících případech:
 - i. Analyzátor musí být schopný detekovat porušení kontraktu, ke kterým skutečně došlo.
 - ii. Analyzátor musí být schopný detekovat porušení kontraktu, ke kterým skutečně nedošlo, ale lze pomocí vektorových hodin odvodit, že k nim může dojít.
 - iii. Analyzátor nesmí detekovat porušení kontraktu v žádném jiném případě (tj. nesmí produkovat *false alarms*).
- 2.7. Analyzátor musí podporovat kontrakty různých typů:
 - i. K jednomu targetu může být zadán jeden spoiler.
 - ii. K jednomu targetu může být zadáno více spoilerů.
 - iii. Targetů může být zadáno v konfiguračním souboru více.
- 2.8. Metody v targetech a spoilerech mohou být:
 - i. bez parametrů,
 - ii. s parametry,
 - iii. s návratovým parametrem.
- 2.9. Metody musí být jednoznačně identifikovány pomocí *fully qualified name*.
- 2.10. Stejně označený parametr v sekvenci metod je považován za shodný parametr.
- 2.11. Stejně označený parametr v targetu a jemu připojeném spoileru musí být považován za shodný parametr.
- 2.12. Parametr zadaný znakem `_` musí být ignorován.

4.2 Přehled částí systému

Pro nový analyzátor je nutné vytvořit několik komponent. Jedná se o samotný analyzátor (*ContractTool*) a jemu přidružený *Cover producer* (konkrétně *ContractCoverProducer*), přičemž obě tyto části se nacházejí v projektu RoadRunner. Dále je nutné vytvořit podporu tohoto nástroje v SearchBestie, kde se o zpracování nasbíraných výsledků bude starat tzv. *Result item creator* (konkrétně *ContractRICreator*). Na diagramu 4.1 je znázorněna komunikace mezi těmito komponentami, přičemž tyto komponenty jsou navrženy tak, aby tato komunikace odpovídala standardní komunikaci mezi SearchBestie a Roadrunnerem prezentované v kapitole 2.6.3.

4.3 Nástroj v Roadrunneru

Nástroj *ContractTool* (dále jen CT) ve Frameworku Roadrunner je hlavní částí analýzy kontraktů, neboť se samotná analýza provádí právě zde. CT rozšiřuje třídu *Tool* frameworku Roadrunner, díky čemuž může zachytávat potřebné události. CT využívá události: *init*, *exit*, *makeShadowVar*, *acquire*, *release*, *create*, *preStart* a *postJoin*. První metoda (tj. *init*) je použita pro inicializaci nástroje, další dvě jmenované metody, *exit* a *makeShadowVar*, jsou důležité pro detekci porušení kontraktu a ostatní metody slouží pro práci s vektorovým časem. CT dále definuje parametr **contractFile**, ve kterém musí být zadána cesta ke konfiguračnímu souboru kontraktů. Takto předaný konfigurační soubor je zpracován právě v metodě *init* za pomoci *Parseru*.



Obrázek 4.1: Konkrétní komunikace mezi komponentami dynamického analyzátoru kontraktů a Searchbestie.

Konfigurační soubor

Parser slouží pro zpracování obsahu konfiguračního souboru, ve kterém se nachází definice kontraktu, a vytvoření ekvivalentní vnitřní reprezentace. Formát konfiguračního souboru vychází z notace použité v definici kontraktů (viz 3.2) a lze jej popsat EBNF¹ gramatikou 4.1.

```

contracts = contract, {new-line, contract} ;
contract = target, "<-", spoilers ;
spoilers = spoiler, {"|", spoiler} ;
target = method-sequence ;
spoiler = method-sequence ;
method-sequence = method, {space, method} ;
method = ret-parameter, method-name, "(", [parameters] ,")" ;
ret-parameter = [parameter, ":"];
parameters = parameter, {"|", parameter} ;
parameter = ( letter-digit, {letter-digit} ) | "_" ;
method-name = letter | {symbol} ;

symbol = letter-digit | "." | "$" ;
letter-digit = letter | digit ;
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
        | "H" | "I" | "J" | "K" | "L" | "M" | "N"
        | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
        | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
        | "c" | "d" | "e" | "f" | "g" | "h" | "i"
        | "j" | "k" | "l" | "m" | "n" | "o" | "p"
        | "q" | "r" | "s" | "t" | "u" | "v" | "w"
        | "x" | "y" | "z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
space = " " ;
new-line = ? ISO 6429 character Carriage Tabulation Return ? ;

```

Kód 4.1: Gramatika generující validní konfigurační soubor.

Příklad 4.2 ukazuje obsah souboru **test1**, který může být vygenerován výše zmíněnou gramatikou a který je jedním ze souborů využitých při testování. V tomto souboru je definován kontrakt, kde targetem je sekvence dvou metod (**m1** a **m2**) a spoilerem je jediná metoda (**m1**).

¹EBNF gramatika je popsána v ISO 14977.

```
cz.vutbr.fit.Test1$Subject.m1() cz.vutbr.fit.Test1$Subject.m2() <-
{ cz.vutbr.fit.Test1$Subject.m1() }
```

Kód 4.2: Příklad konfiguračního souboru.

Takto definovaný konfigurační soubor je zpracován Parserem do vnitřní reprezentace, která je popsána v následující kapitole.

Pro předání cesty ke konfiguračnímu souboru do nástroje Roadrunner je definován nový parametr `-contractFile`, který je přidán jako parametr příkazové řádky do nástroje CT. Tento parametr je při spuštění nástroje CT vždy vyžadován.

4.3.1 Vnitřní reprezentace

Analyzátor si musí uchovat definici kontraktu, která obsahuje definici targetů a jím asociovaných spoilerů. Tyto targety a spoilery jsou používány jako šablony, ze kterých se vytvářejí jejich instance. Targety a spoilery tedy obsahují neměnící se informace, kdežto jejich instance svůj obsah mění. Vztah těchto částí je znázorněn na diagramu 4.2.

Definici kontraktu reprezentuje třída `Contract`, která se v systému nachází vždy pouze jednou. Tato třída obsahuje instance tříd `Target` a `Spoiler`, které dědí od střídy `MethodSequence`. `MethodSequence` obsahuje sekvenci objektů třídy `MethodTemplate`, které reprezentují šablonu metody z definice kontraktu. Tato třída obsahuje důležité informace o metodě, která má být dynamickým analyzátorem sledována. Jedná se o jméno metody, konkrétně ve formátu *fully qualified name*, jména parametrů a jméno návratového parametru². Poslední nezmíněnou třídou je třída `Instance`, které reprezentuje instanci aktuálně rozpracovaného, nebo dokončeného targetu, nebo spoileru. Tato instance se mění při detekování vhodné metody, která je provedena v testovaném programu (viz kapitola 4.3.2). Třída `Instance` obsahuje vektorový čas první a poslední metody, odkaz na šablonu (tj. target nebo spoiler) a jména parametrů ze sekvence metod společně s již přiřazenými hodnotami. Dále instance obsahuje atribut `actualPos`, tj. index metody ze šablony, která byla jako poslední přijata³. Vztah mezi jmény parametrů a jejich hodnotami je více vysvětlen v následujících kapitolách. Zde je důležité zdůraznit, že jména parametrů se nachází v šabloně metod, zatímco jejich hodnoty až v konkrétních instancích.

Počet šablon (tj. targetů a spoilerů) bude neměnný, ovšem počet jejich instancí bude v průběhu analýzy narůstat. Uložení těchto instancí je tedy navrženo tak, aby se mezi nimi dalo co nejrychleji vyhledávat. Instance, které spolu souvisejí, jsou uloženy vždy pohromadě a jsou odděleny od ostatních, které s nimi nesouvisí. K tomuto účelu je využita třída `ShadowVar`, která umožňuje ukládat informace ke každému paměťovému místu (tj. ke každému objektu). Konkrétně je použita instance třídy `ObjectWindows`⁴, která obsahuje přiřazení objektů `Window` ke každému existujícímu vláknu. Instance této třídy je vytvořena při prvním přístupu k paměťovému místu, což je zajištěno v metodě `makeShadowVar`. Třída `Window` je již třídou obsahující jednotlivé instance targetů a spoilerů. Tento vztah je reprezentován na diagramu 4.3, kde jsou pro jednoduchost vynechány položky třídy `Instance`, která je plně zobrazena na diagramu 4.2.

²Jména parametrů nejsou skutečnými jmény parametrů, nýbrž jmény zástupnými, které slouží pro vyjádření vztahů, mezi parametry více metod.

³Atribut `actualPos` ukazuje na poslední z pozorovanou událost ze sekvence metod šablony. Tj. je-li instance vytvořena, je hodnota `actualPos = 0` (ukazuje na první metodu), a je-li instance dokončena, pak je hodnota `actualPos = length - 1`, kde `length` je počet metod v sekvenci dané šablony.

⁴Třída `ObjectWindows` dědí od třídy `ShadowVar`.

4.3.2 Práce s šablonami a instancemi

V dynamickém analyzátoru se vyskytují šablony targetů, spoilerů a jejich instance. V této kapitole je popsáno, jak spolu souvisí a jakým způsobem se s nimi pracuje.

Při inicializaci samotného nástroje jsou vytvořeny všechny targety a spoilery z konfiguračního souboru. Následně je zahájena samotná dynamická analýza, která sleduje důležité události. V kontextu práce s šablonami a instancemi je důležitá pouze událost *exit*, která informuje o ukončení vykonání metody m_{event} . Tato událost obsahuje informace o objektu, na kterém byla vykonána, identifikaci (tj. jméno metody), hodnoty parametrů, se kterými byla zavolána, a návratovou hodnotu této metody. Důvodem, proč je nutné použít metodu *exit* namísto metody *enter*, je ten, že při zavolání metody *enter* jsou k dispozici pouze hodnoty vstupních parametrů, ale nikoli návratová hodnota. Oproti tomu v metodě *exit* tato návratová hodnota již známá je, a proto se v tomto dynamickém analyzátoru kontraktů s parametry zachytávají vykonané metody až po jejich provedení – tedy v metodě *exit*. Jakmile dojde k detekci této události, dynamický analyzátor provede následující kroky:

1. V případě, že metoda m_{event} odpovídá první metodě m_{first} některého targetu, nebo spoileru, vytvoří jeho novou instanci.
2. V případě, že metoda m_{event} odpovídá metodě m_{expect} , která je právě očekávána v některé instanci targetu, nebo spoileru, provede krok *advance* nad touto instancí.
3. Jestliže nedojde ani k jednomu z předchozích případů, je událost ignorována.

4.3.3 Vytvoření nové instance

Jestliže dojde k 1. kroku z předchozí kapitoly 4.3.2, je vytvořena nová instance targetu, nebo spoileru. V tomto kroku je také zmíněno, že metoda m_{event} musí *odpovídat* první metodě m_{first} některého targetu, nebo spoileru. V tomto případě musí dojít k porovnání těchto metod, přičemž při vytváření nové instance ještě nejsou dostupné žádné informace o hodnotách parametrů, a tak je dostačující pouze následující porovnání:

1. Metoda m_{event} musí mít stejně jméno⁵ jako metoda m_{first} .
2. Metoda m_{event} musí mít stejný počet parametrů jako m_{first} .

Na první pohled by se mohlo zdát, že nedochází ke kontrole typů parametrů, a také výstupního parametru. Tato kontrola ovšem není možná, neboť v definici kontraktu se nachází pouze zástupná jména parametrů, nikoli jejich typy.

Při vytvoření nové instance targetu, nebo spoileru je nutné provést následující kroky:

- Přiřadit instanci odkaz na šablonu (target nebo spoiler), ze které je vytvářena.
- Aktuální index v sekvenci metod (*actualPos*) nastavit na hodnotu 0.
- Přiřadit instanci identifikátor vlákna, ve kterém byla metoda m_{event} vyvolána.
- Přiřadit instanci vektorový čas metody m_{event} , který bude značit začátek této instance.
- Vytvořit mapu parametrů *parameters* ze sekvence metod šablony.

⁵K porovnávání dochází nad *fully qualified* jménem metod

- Přiřadit hodnoty parametrů z metody m_{event} k odpovídajícím klíčům mapy $parameters$.

V kapitole 4.3.1 je zmíněno, že zástupné jména parametrů jsou uloženy v šablonách, ale jejich hodnoty v instancích. A právě poslední krok v předcházejícím výčtu se o toto uložení stará. Uložení je realizováno tak, že v každé instanci existuje mapa $parameters$, kde klíče jsou zástupné jména parametrů ze sekvence metod související šablony, a hodnoty jsou při inicializaci nastaveny na hodnotu *Undefined*. Jakmile je detekována metoda m_{event} , jsou hodnoty jejích parametrů přiřazeny k patřičným klíčům v mapě $parameters$. V kontextu vytváření instance lze říci, že hodnoty parametrů z metody m_{event} jsou vždy přiřaditelné. Bude-li ovšem přijata jiná metoda než první metoda v sekvenci šablony, nemusejí být parametry přiřaditelné, a proto je nutné kontrolovat také jejich hodnoty (více v kapitole 4.3.4). Pro lepší pochopení práce s mapou $parameters$ je její inicializace vysvětlena na příkladu 1.

Příklad 1. Necht' existuje target se sekvencí metod: $m_1(X, Y) \quad m_2(Y, Z)$. Necht' je právě detekována událost *exit* s metodou $m_1(5, 7)$. Vzhledem k tomu, že je metoda m_1 první metodou v sekvenci targetu, bude vytvořena nová instance tohoto targetu podle výše uvedeného postupu. Při vytváření instance bude vytvořena mapa $parameters$, která bude mít následující obsah:

```
parameters = {
    "X" = Undefined,
    "Y" = Undefined,
    "Z" = Undefined
}
```

Následně budou do této mapy dosazeny hodnoty parametrů. V šabloně targetu jsou u metody m_1 uvedeny dva parametry se zástupnými jmény X a Y . Takže první hodnota (tj. 5) z detekované metody se přiřadí ke klíči X a druhá hodnota (tj. 7) ke klíči Y . Výsledkem bude mapa $parameters$ s následujícím obsahem:

```
parameters = {
    "X" = 5,
    "Y" = 7,
    "Z" = Undefined
}
```

△

4.3.4 Krok advance

V předchozí kapitole bylo popsáno, jak probíhá vytvoření nové instance targetu, nebo spoileru. Oproti tomu je v této kapitole ukázáno, jako se instance v průběhu dynamické analýzy vyvíjí. Tento případ byl uveden jako 2. krok v kapitole 4.3.2.

Pro zjištění, zda je splněna podmínka z kroku č. 2 z kapitoly 4.3.2, je opět nutné porovnat metody m_{event} a m_{expect} , přičemž toto porovnání navíc obsahuje porovnání hodnot parametrů. Všechny parametry metody m_{event} musejí být *shodné* nebo *přiřaditelné* do parametrů metody m_{expect} , požádmo dané instance. Pokud metoda m_{event} byla volána s určitými hodnotami parametrů, pak platí, že pro všechny její parametry p mohou nastat pouze tyto případy:

1. V mapě $parameters$ nemá parametr p ještě přiřazenu hodnotu.
2. Parametr p má být ignorován.
3. V mapě $parameters$ má parametr p již přiřazenu hodnotu a tato hodnota je totožná s hodnotou parametru p .

4. V mapě *parameters* má parametr *p* již přiřazenu hodnotu a tato hodnota není totožná s hodnotou parametru *p*.

Nastanou-li u všech parametrů metody m_{event} situace z bodu 1, 2 nebo 3, pak jsou parametry považovány za *shodné*. Jestliže jsou navíc splněny obě podmínky z kapitoly 4.3.3, pak je proveden krok *advance* nad danou instancí targetu, nebo spoileru. Případ, kdy nastane situace z bodu 4 je popsán v následujícím příkladu 2.

Příklad 2. Nechť existuje target se sekvencí metod: $m_1(X, Y) \quad m_2(Y, Z)$ z předchozího příkladu 1 a nechť existuje instance tohoto targetu, jejíž atribut *parameters* má následující obsah:

```
parameters = {
    "X" = 5,
    "Y" = 7,
    "Z" = Undefined
}
```

Nechť je detekována událost *exit* s metodou $m_2(9, 1)$. Tato událost je očekávanou metodou v existující instanci targetu a dojde k porovnání parametrů. V šabloně targetu jsou u metody m_2 uvedeny dva parametry se zástupnými jmény *Y* a *Z*. První hodnota (tj. 9) z detekované metody m_2 se přiřadí ke klíči *Y*, což ale není možné, jelikož tento parametr již má dosazenu jinou hodnotu. Tato událost tedy nemůže být další události v sekvenci metod této instance.

△

Krok *advance* umožňuje instanci *i* posunout se vpřed k jejímu dokončení. Je-li metoda m_{event} shodná s metodou m_{expect} (tj. jsou splněny výše uvedené podmínky), pak je nad danou instancí *i* proveden jeden z následujících kroků:

- Je-li m_{expect} volána s parametry, z nichž všechny již mají přiřazenu totožnou hodnotu v mapě *parameters*, pak dojde k inkrementaci hodnoty atributu *actualPos* o hodnotu 1.
- Je-li m_{expect} volána s parametry, z nichž alespoň jeden nemá přiřazenu hodnotu v mapě *parameters*, pak dojde k vytvoření kopie i_{new} dané instance *i*. Této nové instanci i_{new} je přiřazen odkaz na rodičovskou instanci *i*, inkrementována hodnota atributu *actualPos* o hodnotu 1 a aktualizována mapa *parameters* o nově získané hodnoty parametrů.

Důvodem, proč se ve druhém případě vytváří nová instance, je právě dosazení parametrů. Tato situace je vysvětlena na následujícím příkladu 3.

Příklad 3. Nechť je zadána definice kontraktu následovně:

```
m1(X)  m2(Y) <- { m3(X, Y) }
```

Pak se dynamický analyzátor bude snažit nalézt target se sekvencí metod *m1*, *m2*, které jsou volány s parametry *X* a *Y*. Zároveň se bude snažit nalézt spoiler, který obsahuje pouze samotnou metodu *m3*, která je volána s totožnými hodnotami *X* a *Y*. Na diagramu 4.4 je znázorněna situace, kdy je volána metoda *m2* dvakrát, pokaždé s jiným parametrem, a až druhé volání této metody vede k vytvoření targetu, který může být porušen zadáným spoilerem.

Pro popis situace je zavedeno následující označení: *m1(0)* značí vykonání metody *m1*, s parametrem *0*, *Instance{T; MS: m1; PAR: X=0, Y=Undefined}* značí instanci targetu (*T*) s detekovanou sekvencí metod (*MS*) a parametry (*PAR*).

V tuto chvíli bude popsáno pouze vytváření instancí targetu, nikoli spoileru, protože instance spoileru bude jediná: *Instance{S; MS: m3; PAR: X=1, Y=2}*. Instance targetu budou vytvářeny/upravovány na základě detekovaných metod následovně:

1. Detekována metoda: `m1(1)`
`Instance{T; MS: m1; PAR: X=1, Y=Undefined}`
2. Detekována metoda: `m2(0)`
`Instance{T; MS: m1; PAR: X=1, Y=Undefined},`
`Instance{T; MS: m1, m2; PAR: X=1, Y=0}`
3. Detekována metoda: `m2(2)`
`Instance{T; MS: m1; PAR: X=1, Y=Undefined},`
`Instance{T; MS: m1, m2; PAR: X=1, Y=0},`
`Instance{T; MS: m1, m2; PAR: X=1, Y=2}`

V 1. kroku je vytvořena nová instance targetu. Ve 2. kroku je viditelné, že nedojde pouze ke kroku *advance* nad existující instancí, ale je nejprve vytvořena kopie této instance, a až poté je vykonán krok *advance* nad touto kopií. Tento krok je důležitý, protože nově vytvořená instance neodpovídá instanci spoileru, neboť má dosazeny jiné hodnoty parametrů, než se vyskytují ve spoileru. Když se následně v kroku 3 detekuje vykonání metody `m2` s jiným (tj. správným) parametrem, dojde opět k vytvoření kopie původní instance z kroku 1 a k provedení kroku *advance* nad touto kopií. Tato nově vytvořená instance již odpovídá spoileru. Pokud by tedy nedošlo k vytvoření kopie instance v kroku 2 a pouze by se provedl krok *advance*, pak by nebylo možné vytvořit správnou instanci ve 3. kroku.

△

Jestliže došlo ke kroku *advance*, je nutné zkontolovat, zda instance není dokončena. Tento případ nastane, jestliže je hodnota atributu *actualPos* o 1 menší, než počet metod v sekvenci šablony. Je-li instance dokončena, pak je již nejprve přiřazen vektorový čas ukončení, kterým je čas poslední události m_{event} . Následně jsou hledány související dokončené protějšky⁶ a konečně, dojde-li k případu, kdy je nalezen alespoň jeden protějšek, pak je na základě vektorových hodin zkontolováno, zda jsou tyto dvě instance souběžné. Platí-li i poslední podmínka, došlo k detekci porušení kontraktu.

4.3.5 Vektorový čas

Tato kapitola popisuje, jakým způsobem je navržena práce s vektorovými hodinami, a které metody jsou pro tyto hodiny důležité. Vektorové hodiny jsou důležité pro určení, zda jsou dvě instance targetu a spoileru souběžné, tj. zda se mohou vyskytnout v takovém proložení, aby došlo k jejich porušení (více viz 3.2).

Princip práce s vektorovým časem, který je použitý v dynamickém analyzátoru kontraktů, vychází ze článku [19]. Tato implementace byla vybrána z toho důvodu, aby byla shodná s implementací dynamického analyzátoru kontraktů v jazyce C/C++ v projektu ANaConDA. Vektorový čas je udržován v každém vlákně a objektu, který je použitý jako zámek. V nástroji Roadrunner jsou pro jeho použití využity následující metody:

Create V této metodě dochází k vytvoření instance vektorových hodin pro nové vlákno.

PreStart V této metodě dochází k inkrementaci vektorových hodin obou vláken, tj. rodičovského i nově vytvořeného vlákna.

⁶Protějškem je myšlen druhý z dvojice target-spoiler. Tj. dojde-li k ukončení instance targetu, pak jsou hledány související ukončené spoilery; dojde-li k ukončení instance spoileru, pak jsou hledány související ukončené targety.

Acquire Zde je aktualizován vektorový čas vlákna pomocí vektorového času zámku⁷.

Release Zde je aktualizován vektorový čas zámku pomocí vektorového času vlákna. Dále je inkrementován čas vlákna v jeho vektorových hodinách.

PostJoin V této metodě dochází k aktualizaci hodin rodičovského vlákna pomocí vektorového času ukončovaného vlákna. Dále proběhne inkrementace vektorových hodin rodičovského vlákna.

4.3.6 Informace o detekci porušení

Dojde-li k detekování porušení kontraktu, je třeba poskytnout uživateli co nejvíce informací o tomto porušení, které budou následně vytiskeny na chybový výstup. Jako důležité informace byly vybrány následující:

- Metoda, po které došlo k detekci porušení.
- Vlákno, ve kterém byla metoda vyvolána.
- Stack trace vlákna, ve kterém byla metoda vyvolána.
- Instance targetu, který byl porušen.
- Instance spoileru, který jej porušil.

4.4 Podpora nástroje v SearchBestie

Tato kapitola popisuje návrh částí systému, díky kterým je možné dynamický analyzátor kontraktů používat z platformy SearchBestie. První částí je vytvoření tzv. *cover produceru* (dále jen CP) pro analyzátor kontraktů. Další částí je úprava nástroje *RRPlugin* a poslední částí je vytvoření tzv. *result item creatoru* (dále jen RIC)⁸. Kromě vytvoření těchto částí je třeba navrhnout reprezentaci parametrů pro analyzátor kontraktů, které budou předány v konfiguračním souboru pro platformu SearchBestie.

4.4.1 Cover producer a úprava nástroje RRPlugin

CP pro analyzátor kontraktů (nazvaný *Contract cover producer*, dále jen CCP) má za úkol sbírat informace o porušení kontraktů a po skončení testování tyto informace uložit do výstupního souboru. CCP získává informace o detekovaných porušených přímo z nástroje CT. K reportování chyb ovšem dochází pouze v případě, že je analyzátor spouštěn z prostředí SearchBestie, tj. kromě nástroje CT je v řetězci nástrojů také RRPlugin, který vytváří instanci této třídy CCP. Pokud RRPlugin není v řetězci nástrojů při spuštění RoadRunneru, nedojde k vytvoření instance CCP a CT vypisuje nalezené chyby pouze na chybový výstup. Tato třída je stejně jako CT součástí projektu Roadrunner.

Struktura CCP je znázorněna na diagramu 4.6. Důležité jsou především metody *reportContractViolation* a *writeResultToFile*. První zmíněná metoda slouží pro zaznamenání porušení kontraktu a druhá metoda slouží pro vytvoření výstupního souboru a zapsání výsledku. CCP si zaznamenává počet porušení jednotlivých targetů, které jsou identifikované pomocí přiděleného identifikátoru. Obsah výstupního souboru je navržen jako posloupnost čísel oddělených dvojtečkou:

$$x_0 : x_1 : \dots : x_{n-1} \quad (4.1)$$

⁷Detailnější popis aktualizací vektorových časů je popsán ve zmíněném článku [19].

⁸Komunikace mezi těmito částmi byla popsána v kapitole 4.2.

kde x_i značí počet nalezených porušení targetu s identifikátorem i a n značí počet všech targetů v kontraktu. Takto formátovaný výstupní soubor je později zpracován třídou *ContractRICreator*, která je popsána v kapitole 4.4.2.

Instance CCP je vytvářena v nástroji RRPlugin tak, jako ostatní CP. Z tohoto důvodu musí být CCP také korektně ukončen v momentě, kdy dochází k ukončení nástroje RRPlugin. Proces vytvoření, běhu a ukončení CCP je znázorněn na diagramu 4.1.

4.4.2 ContractRICreator a úprava SearchBestie

Pro podporu analyzátoru kontraktů v platformě SearchBestie je nejprve třeba definovat nové parametry do vstupního XML konfiguračního souboru. Těmito novými parametry jsou:

- `<parameter key="contractFile"/>` – Parametr definující cestu ke konfiguračnímu souboru kontraktu.
- `<parameter key="CTcontract"/>` – Parametr povolující vytvoření CCP.

Tyto parametry jsou vnořeny do elementu `parameters`, díky čemuž jsou automaticky zpracovány. Parametry, jejichž klíč začíná prefixem *CT*, jsou považovány za parametry povolující jednotlivé CP a nejsou předávány RoadRunneru. Ostatní parametry jsou beze změny Roadrunneru předávány, a proto musí být klíč nového parametry `contractFile` shodný jako parametr definovaný v kapitole 4.2. Pro parametr `CTcontract` je ještě nutné přidat hodnotu výčtového typu `CONTRACT(...)` do třídy *ConcurrencyCoverage*⁹. Díky tomuto kroku je mimo jiné možné upravit fitness funkci, která se používá pro ohodnocení jednoho běhu testu například takto:

```
<fitness class="cz.vutbr.fit.sbestie.search.fitness.FitnessExpression"
    name="MyFitness">
    <parameters>
        <parameter key="expression" value="CONTRACT" />
    </parameters>
</fitness>
```

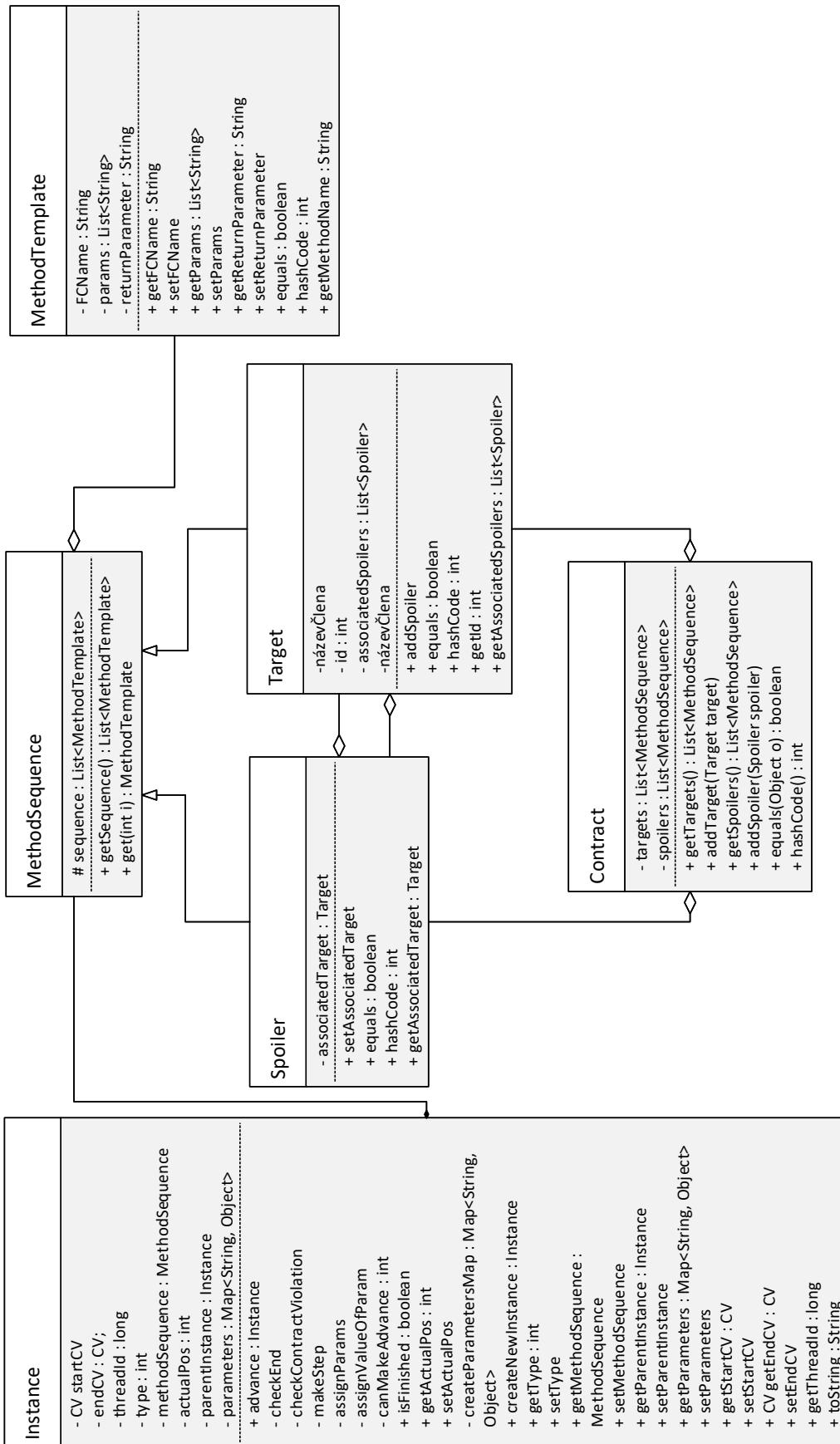
Takto definovaná fitness funkce se bude snažit maximalizovat počet nalezených porušení kontraktů.

Posledním krokem je vytvoření tzv. *Contract result item creatoru* (dále jen CRIC). Tato třída slouží ke zpracování výsledků z RoadRunneru (konkrétně z CCR) a vytvoření výsledku, se kterým dokáže pracovat SearchBestie. Jako tento výsledek byla použita třída *IntResultItem*, která reprezentuje výsledek jednoho testu pouze číselnou hodnotou. CRIC tedy zpracuje sekvenci hodnot reprezentující počty nalezených porušení jednotlivých targetů, tyto výsledky seče a vytvoří instanci třídy *IntResultItem*, do které uloží vypočítanou sumu. Tento výsledek bude reprezentovat počet všech nalezených porušení kontraktu, čehož je využito ve fitness funkci definované v předchozí kapitole.

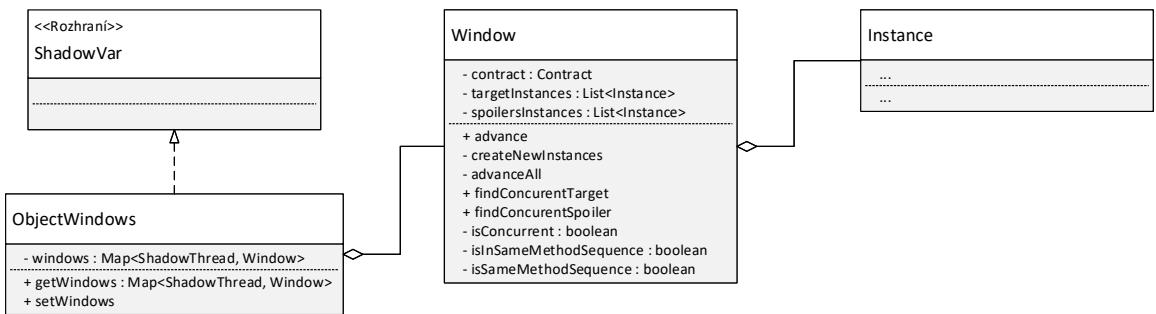
Na diagramu je znázorněna struktura třídy *ContractResultItemCreator*, která rozšiřuje třídu třídu *ResultItemCreator*, a je tedy nutné zářadit výše popsanou funkcionalitu do správných metod této nadřídy. Důležité jsou zejména metody *acceptEvent* a *afterTest*. První jmenovaná metoda slouží pro uložení názvu podsložky¹⁰ a identifikátoru testu, který byl spuštěn. Druhá metoda je zavolána po skončení testu a je využita pro zpracování výstupního souboru z RoadRunneru. Výstupem této metody je číselný výsledek testu, reprezentovaný instancí třídy *IntResultItem*.

⁹Tato hodnota definuje prefixy pro výstupní soubory, názvy složek atd.

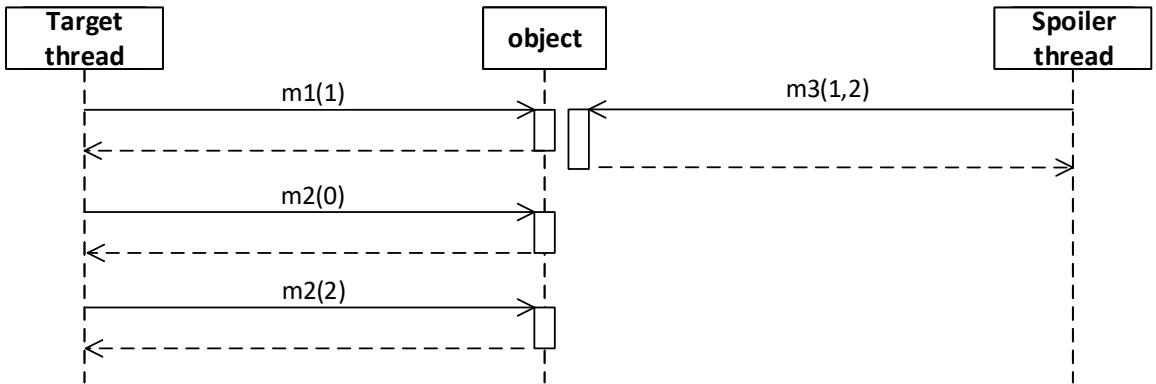
¹⁰V případě hledání porušení kontraktů je podsložka vždy `contract/`. Název této složky je definován ve výčtu třídy *ConcurrencyCoverage* zmíněném výše.



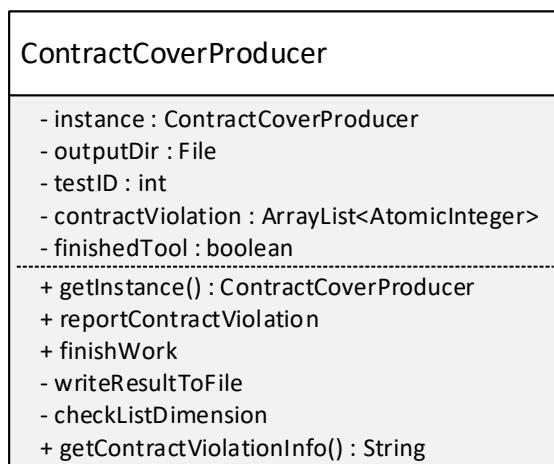
Obrázek 4.2: Diagram zobrazující třídy *Target*, *Spoiler*, *Instance* a jím přidružené třídy.



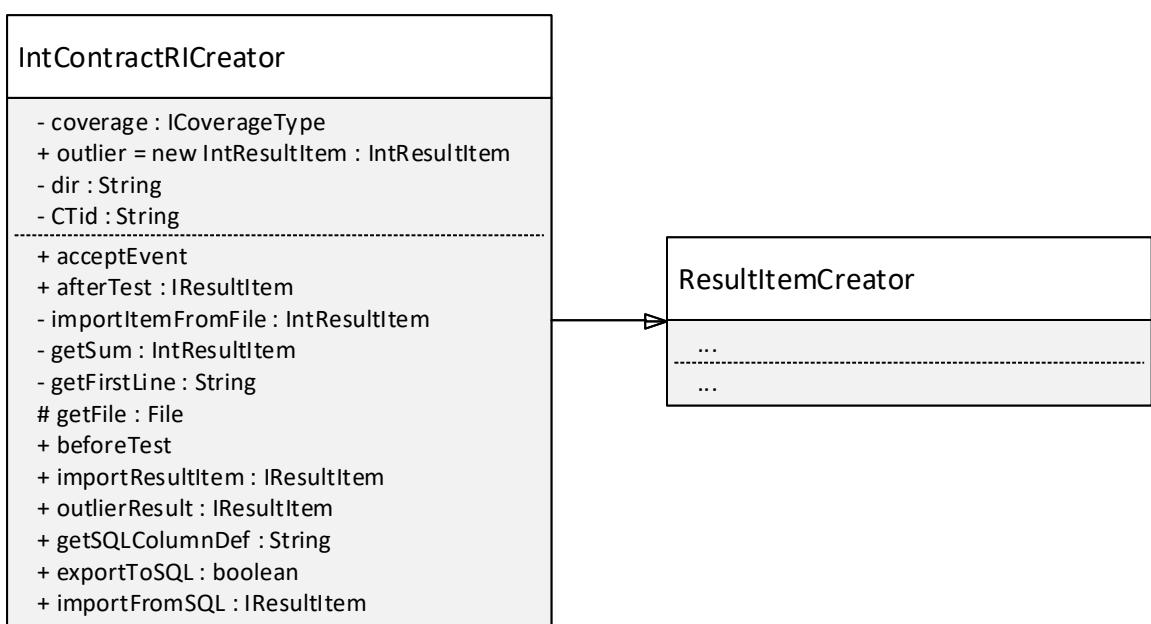
Obrázek 4.3: Diagram zobrazující třídu ObjectWindows a jí přidružené třídy.



Obrázek 4.4: Příklad volání metod nad stejným objektem s různými parametry.



Obrázek 4.5: Diagram zobrazující třídu *ContractCoverProducer*.



Obrázek 4.6: Diagram zobrazující třídu *ContractResultItemCreator*.

Kapitola 5

Implementace analyzátoru kontraktů

Tato kapitola popisuje implementaci analyzátoru kontraktů v nástroji Roadrunner, jeho propojení s platformou SearchBestie a řešení problémů, které při implementaci vznikly. Všechny části byly implementovány v jazyce Java, jelikož je v tomto jazyku napsán jak celý framework Roadrunner, tak SearchBestie.

5.1 Dynamický analyzátor kontraktů

Dynamický analyzátor byl implementován podle návrhu v kapitole 4.3. Zdrojové soubory tohoto analyzátoru se nacházejí v adresáři `src/tools/contract`, testovací programy v adresáři `.test/src/cz/vutbr/fit/` a konfigurační soubory k těmto programům v adresáři `.test/contract-config`.

5.1.1 Spuštění

Před spuštěním je nutné nejprve nastavit framework RoadRunner dle pokynů v příloze A. Spuštění samotného nástroje je potom možné pomocí příkazu:

```
rrrun -classpath=.test/src -tool=CT -contractFile=path/ TestProgram
```

kde:

- `path/` udává cestu ke konfiguračnímu souboru konaktu,
- `TestProgram` udává třídu testovaného programu obsahující funkci `main`¹.
- `CT` v parametru `tool` značí `ContractTool`, tedy analyzátor kontraktů.

Konkrétní příklad spuštění může být demonstrován na jednom z testovacích programů:

```
rrrun -classpath=.test/src -tool=CT -contractFile=.test/contract-config/test11
      cz.vutbr.fit.Test11
```

Pokud dojde k nalezení porušení konaktu, pak je na chybový výstup vytisknuto toto hlášení:

```
##
## =====
## ContractTool Error
##
##     Thread: 2
```

¹Třída musí být před spuštěním přeložena.

```

##          Blame: cz/vutbr/fit/Test11$Subject.m3() V
##          Count: 1      (max: 100)
##          ERROR: Contract violation.
##          Target: : Instance{m1,m2,m3}
##          Spoiler: : Instance{m2,m3}
##          Stack trace: Use -stacks to show stacks...
## =====
##
```

Hlášení může být rozšířeno o stack trace vlákna, které toto porušení způsobilo. Ke zobrazení stack trace slouží parametr `-stacks`.

5.2 Propojení analyzátoru se SearchBestie

Propojení analyzátoru a SearchBestie bylo implementováno podle návrhu v kapitole 4.4.

Zdrojové soubory související s podporou pro analyzátor kontraktů jsou umístěny v následujících adresářích:

- Soubory v projektu SearchBestie:
 - `IntContractRiCreator`: `src/czvutbr/fit/sbestie/storage/resultitem/`,
 - `ConcurrencyCoverage`: `src/cz/vutbr/fit/sbestie/instrument/coverage/`,
 - `RoadRunnerTest`: `src/cz/vutbr/fit/sbestie/instrument/rrunner/`.
- Soubory v projektu Roadrunner: `.test/contract-config`.

5.2.1 Spuštění

Před spuštěním je nutné nejprve nastavit SearchBestie dle pokynů v příloze A. Jako parametr `-config` je předávána cesta ke konfiguračnímu souboru. Pro spuštění testování pomocí dynamického analyzátoru kontraktů je třeba, aby tento konfigurační soubor obsahoval nástroj *ContractTool* v řetězci nástrojů, tj. parametr `tools` musí obsahovat řetězec CT. Dále je třeba nastavit parametr `CTcontract` na hodnotu `true`, která způsobí vytvoření CP určeného pro analyzátor kontraktů. Souhrnně tedy musí být do konfiguračního souboru přidány následující dva řádky:

```

<parameter key="tools" value="CT" />
<parameter key="CTcontract" value="true" />
```

Ostatní parametry konfiguračního souboru jsou popsány v kapitole 2.6.2 a lze je libovolně nastavovat.

5.3 Implementační problémy

Při návrhu dynamického analyzátoru ve frameworku RoadRunner bylo předpokládáno, že v metodách *enter* a *exit* jsou k dispozici hodnoty parametrů, se kterými byly metody zavolány, případně také návratová hodnota metody. V průběhu implementace bylo zjištěno, že objekty, které jsou předány do zmíněných metod, obsahují pouze statické informace o těchto parametrech, tj. počet parametrů a jejich datové typy. Bylo tedy nutné upravit RoadRunner tak, aby hodnoty parametrů byly v metodách *enter* a *exit* k dispozici. Úpravy, které byly provedeny jsou popsány v následujících kapitolách 5.3.1 a 5.3.2.

5.3.1 Rozšíření RoadRunneru o parametry metod

RoadRunner reprezentuje událost vstupu (i výstupu) objektem třídy *MethodEvent*. Instance této třídy je předána jako parametr jak do metody *enter*, tak i do metody *exit* třídy *Tool*. Je tedy nutné rozšířit třídu *MethodEvent* o atribut, který obsahuje hodnoty parametrů. Tento parametr byl nazván *params* a je typu *Object []*. Důvod, proč byl zvolen tento typ je, že v Javě se mohou vyskytovat jak primitivní, tak referenční datové typy. Všechny referenční datové typy rozšiřují třídu *Object*, takže mohou být v tomto poli uloženy. Primitivní datové typy třídu *Object* nerozšiřují, nicméně ke každému primitivnímu datovému typu existuje referenční datový typ. Převod primitivního datového typu na referenční se provádí pomocí metody *valueOf* daného referenčního typu. Pokud tedy mají být uloženy všechny datové typy do jednoho pole, je nutné všechny typy převést na referenční a tyto referenční datové typy uložit do tohoto pole. Výsledkem tedy je pole, obsahující jak původní referenční datové typy, tak původně primitivní datové typy.

Objekty třídy *MethodEvent* jsou generovány ve třídě *RREventGenerator*, konkrétně v metodě *enter*. Do této metody je nutné přidat další parametr *Object [] params*, ve kterém budou zaslány hodnoty parametrů. Uvnitř této metody probíhá vytvoření instance třídy *MethodEvent*, které je třeba přiřadit hodnoty parametrů z parametru *params*. Tato metoda je vyvolávána staticky dle její deklarace ve třídě *Constants*. Tento záZNAM je tedy třeba modifikovat a přidat parametr *params*. Vytvořená instance třídy *MethodEvent* je předána prvnímu nástroji v řetězci nástrojů v metodě *enter*. Zjednodušená upravená metoda *enter* třídy *RREventGenerator* je zobrazena v kódu 5.1.

```
1 public static void enter(final Object target, final int methodDataId,
2                           final ShadowThread td, Object[] params) {
3     ...
4     final MethodEvent me = td.enter(target, methodData);
5     me.setParams(params);
6     ...
7     firstTool.enter(me);
8 }
```

Kód 5.1: Upravená metoda *enter* třídy *RREventGenerator*.

Dalším krokem je získání hodnot parametrů, se kterými byla zavolána metoda v testovaném programu. K tomuto kroku je nutné nejprve uvést, jakým způsobem Roadrunner získává informace o testovaném programu. RoadRunner využívá knihovnu ASM², která slouží pro manipulaci a analýzu Java byte kódu. Konkrétně se analýza a manipulace s metodami provádí ve třídě *SyncAndMethodThunkInserter* v metodě *createMethodThunk*, kde je dostupný kontext vláken testovaného programu (tj. zásobník, registry, atd.) prostřednictvím objektu *mv* typu *MethodVisitor*³. Odtud je dále volána metoda *enter* třídy *RREventGenerator*, přičemž k jejímu volání dochází ihned po vstupu do metody v testovaném programu. Metoda *createMethodThunk* třídy *SyncAndMethodThunkInserter* byla původně volána způsobem zobrazeným v kódu 5.2.

```
1 private void createMethodThunk(int access, String name, String desc,
2                                 String signature, String[] exceptions, String wrappedMethodName, int
3                                 maxLocals) {
4     ...
5     MethodInfo m = method;
6     mv.push(m.getId());
7     mv.invokeStatic(Constants.THREAD_STATE_TYPE,
8                      Constants.CURRENT_THREAD_METHOD);
9     mv.invokeStatic(Constants.MANAGER_TYPE, Constants.ENTER_METHOD);
```

²Knihovna ASM je dostupná na adrese <http://asm.ow2.org/>.

³MethodVisitor je třída z knihovny ASM, sloužící k analýze a manipulaci s byte kódem souvisejícím s metodami.

```
7     ...
8 }
```

Kód 5.2: Upravená metoda *enter* třídy *RREventGenerator*.

Ve výše uvedeném kódu proběhne na řádku 4 uložení methodDataId na zásobník. Na řádku 5 dojde k vyvolání statické metody *getCurrentShadowThread*, definované ve třídě Constants, která uloží na vrchol zásobníku objekt ShadowThread asociovaný k aktuálnímu vláknu. Jako poslední proběhne na řádku 6 vyvolání statické metody *enter* třídy *RREventGenerator*. Pro správné vyvolání této metody, je nutné mít umístěny všechny její parametry na zásobníku, což ale není splněno, jelikož byl do této metody přidán parametr *Object[] params*. Z tohoto důvodu je mezi 5. a 6. řádek programu vložen kód, který na zásobník umístí pole obsahující hodnoty parametrů metody z testovaného programu. Všechny tyto operace jsou prováděny pomocí byte kódových instrukcí, které jsou volány pomocí metod objektu třídy *MethodVisitor*. Například instrukce *bipush* se vykoná provedením metody *mv.visitIntInsn(Opcodes.BIPUSH, i)*⁴. Upravená metoda *createMethodThunk* je zobrazena a vysvětlena v kódu 5.3.

```
1 private void createMethodThunk(int access, String name, String desc,
2                               String signature, String[] exceptions, String wrappedMethodName, int
3                               maxLocals) {
4     ...
5     MethodInfo m = method;
6     mv.push(m.getId());
7     mv.invokeStatic(Constants.THREAD_STATE_TYPE,
8                      Constants.CURRENT_THREAD_METHOD);
9
10    ///////////////////////////////////////////////////////////////////
11    // ZACATEK: vlozeny kod vytvarejici pole hodnot parametru
12    ///////////////////////////////////////////////////////////////////
13
14    Type[] paramTypes = Type.getArgumentTypes(method.getDescriptor());
15    int paramLength = paramTypes.length;
16
17    if(paramLength > 0) {
18
19        // ulozeni velikosti pole parametru na zasobnik
20        mv.visitIntInsn(Opcodes.BIPUSH, paramLength);
21
22        // vytvoreni pole params o~velikosti umistene na vrcholu
23        // zasobniku
24        mv.visitTypeInsn(Opcodes.ANEWARRAY, "java/lang/Object");
25
26        // ulozeni tohoto pole do locals
27        mv.visitVarInsn(Opcodes.ASTORE, paramLength +
28                        PARAM_OFFSET);
29
30        Integer i = new Integer(0);
31
32        // cyklus pres vsechny parametry dle jejich typu
33        for (Type type : paramTypes) {
34
35            // ulozeni pole params na vrchol zasobniku
36            mv.visitVarInsn(Opcodes.ALOAD, paramLength +
37                            PARAM_OFFSET);
```

⁴Seznam všech instrukcí je dostupný na <https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html>

```

33          // ulozeni idenxu na vrchol zasobniku
34          mv.visitIntInsn(Opcodes.BIPUSH, i);
35
36          // zapouzdreni primitivnich typu a
37          // zduplikovani hodnoty i-teho parametru
38          // na vrchol zasobniku
39          i = boxAndVisitVariable(mv, i, type);
40
41          // ulozeni hodnoty parametru do pole params
42          // na index i
43          mv.visitInsn(Opcodes.AASTORE);
44          i++;
45      }
46
47      // ulozeni pole params na vrchol zasobniku
48      mv.visitVarInsn(Opcodes.ALOAD, paramLength +
        PARAM_OFFSET);
49  } else {
50      // ulozeni hodnoy NULL na vrchol zasobniku
51      mv.visitInsn(Opcodes.ACONST_NULL);
52  }
53
54  /////////////////////////////////
55  // KONEC: vlozeny kod vytvarejici pole hodnot parametru
56  /////////////////////////////////
57
58  mv.invokeStatic(Constants.MANAGER_TYPE, Constants.ENTER_METHOD);
59  ...
60 }
```

Kód 5.3: Upravená metoda `createMethodThunk` zajišťující vytvoření pole hodnot parametrů volané metody.

V kódu výše je metoda `boxAndVisitVariable`, která provádí zapouzdření primitivních typů do referenčních datových typů. Hodnoty parametrů metody v testovaném programu jsou umístěny v *locals* a proto tento převod musí probíhat opět na úrovni Java byte kódu. Kód této metody je znázorněn v příkladu 5.4. Jestliže hodnota parametru na indexu *i* je primitivního typu, pak tento kód načte tuto hodnotu z *locals* a uloží ji na vrchol zásobníku. Poté zavolá metodu `valueOf` odpovídajícího datového typu, která vytvoří hodnotu referenčního datového typu z hodnoty na vrcholu zásobníku a výsledek umístí opět na vrchol zásobníku. Pokud již parametr je referenčního datového typu, pak je pouze načten z *locals* na vrchol zásobníku.

```

1  private Integer boxAndVisitVariable(IRRMethodAdapter mv, Integer i, Type
2      type) {
3          if (type.equals(Type.BOOLEAN_TYPE)) {
4              mv.visitVarInsn(Opcodes.ILOAD, i + 1);
5              mv.visitMethodInsn(Opcodes.INVOKESTATIC,
6                  "java/lang/Boolean", "valueOf",
7                  "(Z)Ljava/lang/Boolean;", false);
8          } else if (type.equals(Type.BYTE_TYPE)) {
9              mv.visitVarInsn(Opcodes.ILOAD, i + 1);
10             mv.visitMethodInsn(Opcodes.INVOKESTATIC,
11                 "java/lang/Byte", "valueOf",
12                 "(B)Ljava/lang/Byte;", false);
13         } else if (type.equals(Type.CHAR_TYPE)) {
14             mv.visitVarInsn(Opcodes.ILOAD, i + 1);
15             mv.visitMethodInsn(Opcodes.INVOKESTATIC,
16                 "java/lang/Character", "valueOf",
17                 "(C)Ljava/lang/Character;", false);
18     }
```

```

11         } else if (type.equals(Type.SHORT_TYPE)) {
12             mv.visitVarInsn(Opcodes.ILOAD, i + 1);
13             mv.visitMethodInsn(Opcodes.INVOKESTATIC,
14                 "java/lang/Short", "valueOf",
15                 "(S)Ljava/lang/Short;", false);
16         } else if (type.equals(Type.INT_TYPE)) {
17             mv.visitVarInsn(Opcodes.ILOAD, i + 1);
18             mv.visitMethodInsn(Opcodes.INVOKESTATIC,
19                 "java/lang/Integer", "valueOf",
20                 "(I)Ljava/lang/Integer;", false);
21         } else if (type.equals(Type.LONG_TYPE)) {
22             mv.visitVarInsn(Opcodes.LLOAD, i + 1);
23             mv.visitMethodInsn(Opcodes.INVOKESTATIC,
24                 "java/lang/Long", "valueOf",
25                 "(J)Ljava/lang/Long;", false);
26             i++;
27         } else if (type.equals(Type.FLOAT_TYPE)) {
28             mv.visitVarInsn(Opcodes.FLOAD, i + 1);
29             mv.visitMethodInsn(Opcodes.INVOKESTATIC,
30                 "java/lang/Float", "valueOf",
31                 "(F)Ljava/lang/Float;", false);
32         } else if (type.equals(Type.DOUBLE_TYPE)) {
33             mv.visitVarInsn(Opcodes.DLOAD, i + 1);
34             mv.visitMethodInsn(Opcodes.INVOKESTATIC,
35                 "java/lang/Double", "valueOf",
36                 "(D)Ljava/lang/Double;", false);
37             i++;
38         } else {
39             mv.visitVarInsn(Opcodes.ALOAD, i + 1);
40         }
41     }
42     return i;
43 }
```

Kód 5.4: Metoda zajišťující zapouzdření primitivních datových typů.

Výše uvedeným způsobem je docíleno zduplikování hodnot parametrů volané metody a jejich postupnému předání až do metody *enter* jednotlivých nástrojů. Dalším krokem je obdobným způsobem získat a předat návratovou hodnotu z metod testovaného programu.

5.3.2 Rozšíření RoadRunneru o návratový parametr

Rozšíření o návratovou hodnotu metod je provedeno stejným způsobem jako rozšíření o parametry popsané v předchozí kapitole 5.3.1.

Nejprve je třeba přidat atribut `Object returnValue` do třídy `MethodEvent`. Dalším krokem je přidat stejný parametr do metody *exit* ve třídě `RREventGenerator`, ve které proběhne (stejně jako v metodě *enter* téže třídy) vytvoření instance třídy `MethodEvent` a přiřazení parametru `returnValue` této instanci. Tato metoda je stejně jako metoda *enter* volána z metody `createMethodThunk`, kde musí být návratová hodnota také stejným způsobem získána. Kód získání návratové hodnoty je téměř stejný jako kód získání parametrů. Rozdílem je, že není nutné vytvářet a plnit pole, protože stačí zduplikovat hodnotu na vrcholu zásobníku (návratovou hodnotu) a převést ji na referenční datový typ. Výsledek tohoto převodu je umístěn na vrchol zásobníku před volání metody *exit* třídy `RREventGenerator`, což způsobí předání tohoto parametru této metodě.

5.3.3 Ověření rozšíření

Při implementaci rozšíření z kapitol 5.3.1 a ?? jsem si nebyl jistý, zda je rozšíření vůbec možné, a tak jsem se rozhodl zkонтакtovat pana profesora Stephena N. Freunda, autora projektu RoadRunner, aby mi poradil, jakým způsobem lze rozšíření implementovat. Ten mi nestihl odpovědět hned, nicméně několik dní po dokončení mé implementace mi od něj přišla odpověď, kde mi radil stejný postup, jaký jsem použil, a proto považuji popsaný postup za správný.

Kapitola 6

Ověření funkčnosti

Ověření funkčnosti probíhalo pomocí jednotkového testování v průběhu vývoje, dále byla navržena sada testů pro ověření funkčnosti dynamického analyzátoru kontraktů ve frameworku Roadrunner a následně byl testován vliv spuštění nástroje z prostředí SearchBestie.

6.1 Jednotkové testování

Pro ověřování správnosti některých částí dynamického analyzátoru v RoadRunneru byly v průběhu vývoje používány jednotkové testy (konkrétně *JUnit*). Tyto testy byly použity pro průběžné ověřování funkčnosti jednotlivých částí systému, aby bylo možné odhalit chyby hned při implementaci těchto částí, nikoli až po testování celého nástroje. Jednalo se o testování:

- zpracování konfiguračního souboru,
- porovnávání metod.

6.1.1 Zpracování konfiguračního souboru

Pro zpracování konfiguračního souboru slouží třída *Parser*, přičemž její kontrola je prováděna ve třídě *PraserTest*, kde dochází ke kontrole zpracování následujících řetězců:

- řetězec parametrů: `X, Y, next , last1,`
- řetězec metody: `method()`,
- řetězec metody s návratovým parametrem: `X:method1()`,
- řetězec více metod: `method1() m2() met3()`,
- řetězec konaktu a spoileru: `X:method1() <- { m2(par,X) }`,
- řetězec složitějšího targetu a více spoilerů `method1() method1() <- { m2() m3() | m4() m5() }`.

Výše uvedené řetězce jsou validními řetězci, které se mohou vyskytovat v definici kontraktů. Testováno je tedy pouze zpracování validních vstupů, nikoli odolnost vůči nesprávným vstupům.

¹Řetězec opravdu obsahuje větší počet mezer mezi parametry z důvodu testování.

6.1.2 Porovnání metod

Druhou částí jednotkového testování je testování porovnání parametrů. Porovnání parametrů je součástí porovnání metod, které je prováděno ve třídě *MethodComparator*. Toto testování je tedy prováděno ve třídě *MethodComparatorTest*, přičemž opět dochází pouze k jednoduchému testování. U parametrů jsou testovány 3 situace:

- Parametry mají definovanou hodnotu v instanci.
- Parametry ještě nemají definovanou hodnotu v instanci.
- Parametry mají být ignorovány.

6.2 Testování dynamického analyzátoru ve frameworku RoadRunner

Pro testování dynamického analyzátoru kontraktů je vytvořeno 46 testů. Tyto testy jsou popsány jednotným způsobem, který je v této kapitole vysvětlen, přičemž samotné testy jsou umístěny v příloze B. Každý test se skládá z textového popisu a sekvenčního diagramu.

Soubory pro tyto testy jsou umístěny ve složce `.test/` v kořenové složce projektu Roadrunner. Spuštění celé sady testů je umožněno pomocí testovacího skriptu `.test/test-big.sh`, přičemž tento script musí být spuštěn z kořenového adresáře `roadrunner/`. Pro každý test čísla X je vytvořen Java soubor `TestX.java`, který obsahuje testovaný program, a konfigurační soubor `testX`. Pro přehlednost je v každém zdrojovém Java souboru uvedena definice konaktu z konfiguračního souboru.

6.2.1 Textový popis testů

Součástí textového popisu testů jsou následující položky:

Kategorie Tato položka udává na co se daný test zaměřuje, přičemž u jednoho testu může být kategorií více. Tato položka slouží pro rychlý přehled mezi všemi testy. Kategorie jsou:

- *JEDNODUCHÝ KONTRAKT* – testování probíhá pouze na jednoduchých kontraktech.
- *SLOŽITĚJŠÍ KONTRAKT* – testování probíhá na složitějších kontraktech.
- *VEKTOROVÝ ČAS* – k odhalení porušení je nutné použítí vektorového času.
- *VÍCE OBJEKTU* – metody targetů a spoilerů jsou volány na více objektů stejných typů.
- *VÍCE SPOILERU* – k jednomu targetu je definováno více spoilerů.
- *VÍCE VLÁKEN* – metody targetů a spoilerů jsou volány z více vláken.
- *IGNOROVÁNÍ OSTATNÍCH METOD* – v programu se nachází volání metod, které je nutné ignorovat.
- *NEDOKONČENÉ SEKVENCE* – v programu se vyskytují nedokončené sekvence metod.
- *PARAMETRY* – kontrakty jsou zadány s parametry.
- *RŮZNÉ TYPY PARAMETRŮ* – v metodách se vyskytují parametry různých typů.
- *IGNOROVÁNÍ PARAMETRŮ* – v konaktu je zadán parametr `_`, který je nutné ignorovat.
- *VÍCE PARAMETRŮ* – v metodách se vyskytuje více parametrů.

- **NÁVRATOVÝ PARAMETR** – v kontraktu je zadán návratový parametr.

Popis Tato položka uvádí detailnější informace o testu a o důvodu porušení (nebo neporušení) konaktu.

Kontrakt Položka kontrakt definuje kontrakt, který je hledán. Definice konaktu je stejného formátu jako obsah konfiguračního souboru dynamického analyzátoru (viz kapitola 4.2).

Očekávaný výsledek Tato položka udává kolik porušení konaktu je očekáváno.

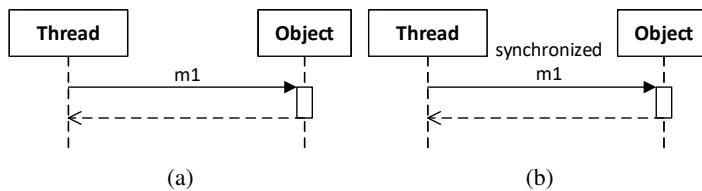
Skutečný výsledek Skutečný výsledek udává, kolik porušení konaktu bylo skutečně detekováno.

Jsou-li položky *očekávaný výsledek* a *skutečný výsledek* shodné, pak se na závěru textového popisu objevuje informace o úspěšném provedení testu.

6.2.2 Sekvenční diagram

Pro lepší znázornění testované situace je každý test doplněn o sekvenční diagram. Tento typ diagramu byl vybrán na základě vědeckého článku [14], kde byly představeny způsoby reprezentace vícevláknových programů v Javě pomocí jazyka UML, a také na základě nejlepšího znázornění požadovaných informací o jednotlivých testech. Přesto lze nalézt informace, jejíž znázornění v tomto diagramu není úplně intuitivní, a proto jsou zde jednotlivé situace popsány.

Použité diagramy znázorňují volání metod objektů z různých vláken. Vyskytují se zde vlákna *Target thread* a *Spoiler thread*, které volají metody na sdílené objekty (*Object*). Na obrázku 6.1a je znázorněno volání metody *m1* objektu *Object* z vlákna *Thread*. Toto volání je vykonáno bez použití synchronizace (tj. bez použití zámku). Oproti tomu obrázek 6.1b znázorňuje volání stejné metody *m1*, ovšem již s použitím synchronizace. Tuto skutečnost znázorňuje klíčové slovo *synchronized*².



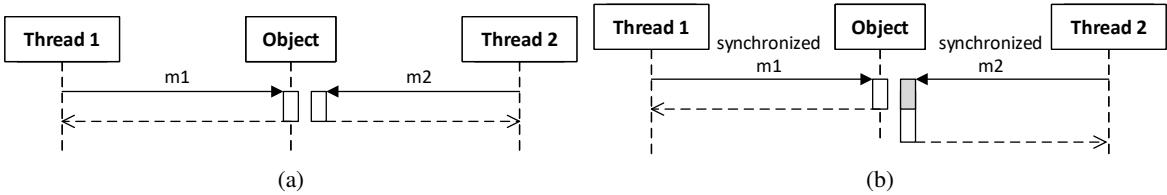
Obrázek 6.1: Volání metody s použitím (a) a bez použití (b) synchronizace.

Na dalším obrázku obrázku 6.2a je znázorněna situace, kdy dvě vlákna (*Thread 1* a *Thread 2*) volají současně dvě metody (*m1* a *m2*), na stejném objektu a bez použití synchronizace. V takovém případě není zřejmé, k jakému proložení mezi těmito voláními dojde. Opačný případ, kdy vlákna volají metody s použitím synchronizace je znázorněn na obrázku 6.2b. V takovémto případě je jasné, že se nejprve provede jedna metoda, a až poté druhá. Na obrázku je tedy znázorněn výlučný přístup k objektu (*Object*)³. Pořadí vykonání metod ovšem není zaručeno, protože toto zobrazení pouze říká, že dojde k postupnému provedení metod. Čekání na uvolnění zámku je znázorněno zašdelou částí u aktivace objektu. Takovýmto způsobem může být voláno i více metod najednou, jak je zobrazeno na obrázku 6.3b. Na obrázku 6.3a je znázorněna situace, kde je z volání metody *m1* navrácena hodnota 5. Pokud dojde k volání více metod a je při tomto volání použita synchronizace, pak je navrácení hodnoty z metody reprezentováno *5=m1()*, který říká, že při volání

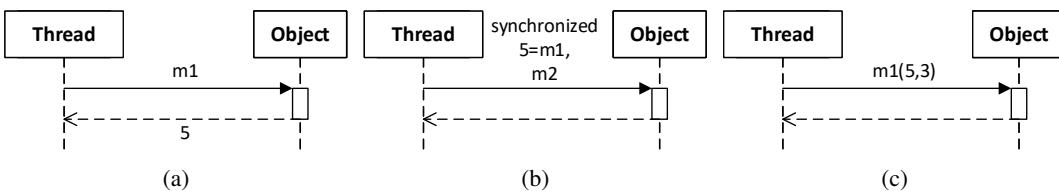
²V příkladech bude uvažováno, že jako zámek je požitý objekt, na kterém jsou volány metody.

³Toto zobrazení odpovídá relaci Happens before (viz kapitola 2.1).

metody $m1$ byla navrácena hodnota 5. Tato reprezentace byla zvolena proto, aby bylo zřejmé, že sekvence metod ($m1$ a $m2$) byla volána s použitím synchronizace. Na posledním obrázku 6.3c je znázorněno volání metod $m1$ s parametry 5 a 3.



Obrázek 6.2: Souběžné (a) a výlučné (b) volání metod stejného objektu.



Obrázek 6.3: Volání metod s návratovou hodnotou (a) a (b) a s parametry (c).

Poslední včí, která se může v diagramu vyskytnout je uspání vláken. Uspání je zobrazeno pomocí volání metody *sleep* vlákna na sebe samé. Parametrem tohoto volání je počet milisekund, na který je vlákno uspáno. Uspání je v testech použito pro skutečnou demonstraci určitého proložení metod. Jestliže se uspání v příkladu nenachází a zároveň jsou metody volány bez synchronizace, pak je zobrazení proložení metod pouze ilustrativní⁴.

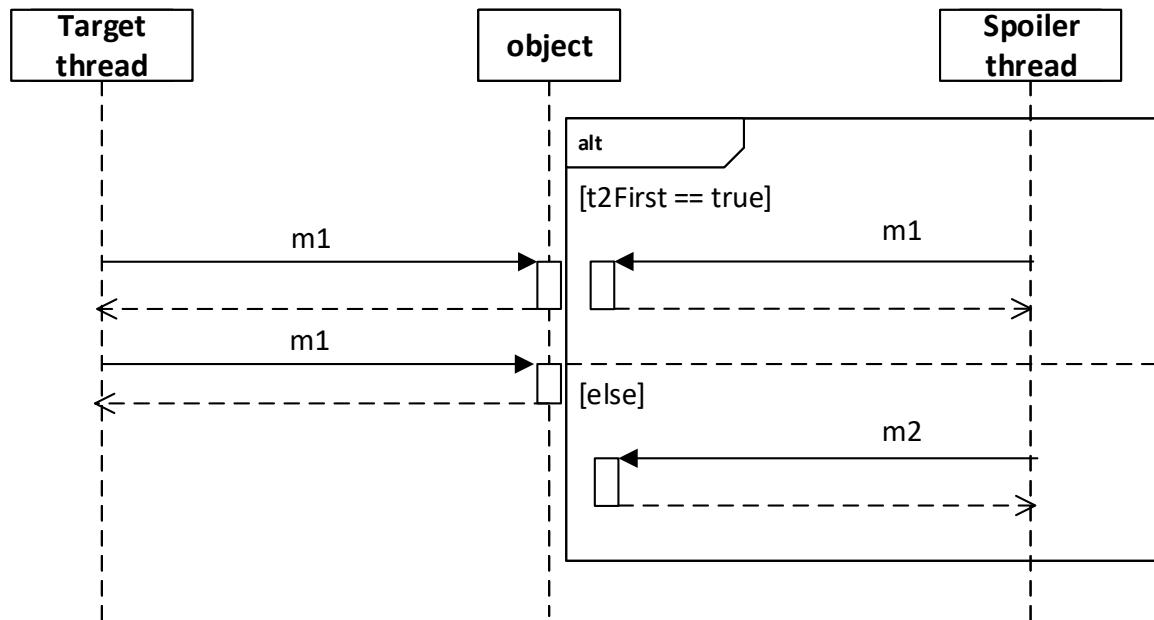
6.3 Testování propojení analyzátoru kontraktů s platformou SearchBestie

Analyzátor kontraktů implementovaný v nástroji RoadRunner dokáže odhalit chyby za pomocí vektorových hodin. Nastávají ovšem případy, kdy ani použití vektorových hodin nedokáže chyby v programu odhalit. Příkladem může být situace na obrázku 6.4. K porušení konaktu dojde, pokud vlákno *Spoiler thread* provede metodu $m1$. To, zda k této situaci dojde, ovlivňuje vlákno *Target thread*. Jestliže se spustí nejdříve toto vlákno, pak nastaví hodnotu konstanty $t2First=false$, a vlákno *Spoiler thread* metodu $m1$ nikdy neprovede. Jelikož je v programu spuštěno první vlákno *Target thread*, tak ve většině případů k porušení konaktu nedojde. Jedním ze způsobů, jakým lze docílit vynucení porušení konaktu, je vkládání šumu, o což se stará právě SearchBestie. Šum je potřeba vložit tak, aby bylo vlákno *Target thread* pozastaveno do té doby, než vlákno *Spoiler thread* provede porovnání (přečtení) proměnné $t2First$. V následujícím testu je tedy porovnáváno spuštění s vkládáním šumu a bez něj. Popis testu vychází ze zavedeného popisu z kapitoly 6.2.

- Kontrakt: $m1() \quad m2() \leftarrow \quad m1()$.
- Nastavení SearchBestie:

⁴Metody mohou být různě proloženy pouze mezi různými vlákny, tj. pořadí volání metod v jednom vlákně zůstává vždy zachováno.

- počet opakování testu: 100,
 - frekvence šumu: 10 %,
 - síla šumu: 500 ms,
 - typ šumu: *yield, sleep, wait, busyWait, syncYields, mixed*.
- Počet porušení bez vkládání šumu: 1 ze 100 spuštění.
 - Počet porušení s vkládáním šumu: 8 ze 100 spuštění.



Obrázek 6.4: Diagram testu 49.

Ve výsledcích testu se objevil 1 případ, kdy došlo k porušení kontraktu bez vkládání šumu, a 8 případů, kdy došlo k detekci porušení kontraktu s vkládáním šumu.

Kapitola 7

Závěr

V této práci jsem navrhl a implementoval dynamický analyzátor kontraktů rozšířený o parametry a návratovou hodnotu. Stěžejní částí bylo vytvoření tohoto analyzátoru jako nástroje v projektu RoadRunner, nicméně nutné také bylo vytvořit propojení tohoto nástroje s platformou SearchBestie. Během implementace bylo zjištěno, že framework RoadRunner neumožňuje získat hodnoty parametrů, se kterými byly metody volány, a bylo tak třeba rozšířit tento nástroj o tuto funkcionality. Rozšíření o hodnoty parametrů bylo implementováno společně s rozšířením o návratové hodnoty z metod tak, že následně i pan profesor Stephen N. Freund, autor projektu RoadRunner, potvrdil správnost tohoto postupu.

Funkčnost analyzátoru kontraktů byla ověřena 46 testy společně s možností spouštět tento nástroj z platformy SearchBestie, která zvyšuje pravděpodobnost nalezení porušení konaktu. Nicméně testování spouštělo SearchBestie pouze se základním nastavením a jednoduchým programem, a nebylo tak možné ověřit, jak dobrého výsledku lze dosáhnout při optimálním nastavení parametrů SearchBestie nad reálným programem. Důvodem bylo zejména to, že se nepodařilo nalézt případ reálného programu v Javě, který by obsahoval porušení nějakého konaktu.

V další práci by bylo vhodné takovýto případ nalézt a ověřit na něm funkčnost tohoto nástroje. Zároveň by bylo třeba optimalizovat tento nástroj na testování velkých programů, neboť na rozdíl od definice dynamického analyzátoru kontraktů v kapitole 3.2.1 toto řešení nezahazuje již nepotřebné instance targetů a spoilerů a při testování velkého programu by mohlo dojít k nedostatku paměti. Toto zahazování instancí nebylo implementováno, jelikož případy, kdy je možné zahazovat tyto instance kontraktů s parametry, ještě nebyly formálně dokázány. Dalším rozšířením by mohlo být vytvoření nového tzv. *Result itemu*, kterým SearchBestie reprezentuje výsledek testu. V současné době se používá pouze číselná hodnota reprezentující počet nalezených porušení napříč všemi targety v konaktu, avšak tento dynamický analyzátor zprostředkovává informace o porušení každého z těchto targetů. SearchBestie by se tak mohla například pokoušet nalézt porušení každého z nich.

Literatura

- [1] Andrews, G. R.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000, ISBN 0-201-35752-6.
- [2] Center, N. A. R.: *JavaTM Pathfinder*. [Online; navštíveno 09.01.2017]. URL <https://http://babelfish.arc.nasa.gov/trac/jpf>
- [3] Edelstein, O.; Farchi, E.; Goldin, E.; aj.: *Framework for testing multi-threaded Java programs*. *Concurrency and Computation: Practice and Experience*, ročník 15, 2003: s. 485–499, doi:10.1002/cpe.654.
URL <http://doi.wiley.com/10.1002/cpe.654>
- [4] Ferreira, C.; andand Diogo G. Sousa, J. L.; Dias, R. J.: *Preventing atomicity violations with contracts*. In *eprint arXiv:1505.02951*, 2015.
URL <https://arxiv.org/abs/1505.02951>
- [5] Ferreira, C.; Fiedor, J.; Lourenco, J.; aj.: *Verifying Concurrent Programs Using Contracts*. To appear In *Proceedings of ICST Conference*, 2017.
- [6] Fiedor, J.; Dudka, V.; Křena, B.; aj.: *Advances in Noise-based Testing of Concurrent Programs*. Software Testing, Verification and Reliability, ročník 25, č. 3, 2015: s. 272–309, ISSN 1099-1689.
URL <http://www.fit.vutbr.cz/research/viewpub.php.cs?id=10275>
- [7] Fiedor, J.; Letko, Z.; Lourenco, J.; aj.: *Dynamic Validation of Contracts in Concurrent Code*. In Proceedings of the 15th International Conference on Computer Aided Systems Theory, *The Universidad de Las Palmas de Gran Canaria*, 2015, ISBN 978-84-606-5438-4, s. 177–178.
URL <http://www.fit.vutbr.cz/research/viewpub.php.cs?id=10817>
- [8] Flanagan, C.; Freund, S. N.: *FastTrack: Efficient and Precise Dynamic Race Detection*. In Proceedings of the 15th International Conference on Computer Aided Systems Theory, ročník 53, *Communications of the ACM*, 2010, s. 93–101, doi:10.1145/1839676.1839699.
URL <http://portal.acm.org/citation.cfm?doid=1839676.1839699>
- [9] Flanagan, C.; Freund, S. N.: *The RoadRunner Dynamic Analysis Framework for Concurrent Programs*. In Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, *ACM New York, USA*, 2010, ISBN 978-1-4503-0082-7, s. 1–8.
URL <https://users.soe.ucsc.edu/~cormac/papers/paste10.pdf>
- [10] Kshemkalyani, A. D.; Singhal, M.: *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge Press, 2011, ISBN 9780521189842.

- [11] Křena, B.; Letko, Z.; Ur, S.; aj.: *A Platform for Search-Based Testing of Concurrent Software*. In PADTAD '10, Proceedings of the 8th Workshop on Parallel and Distributed Systems, Association for Computing Machinery, 2010, ISBN 978-1-60558-823-0, str. 11.
URL <http://www.fit.vutbr.cz/research/viewpub.php?id=9275>
- [12] Křena, B.; Letko, Z.; Vojnar, T.: *Noise Injection Heuristics for Concurrency Testing*. Lecture Notes in Computer Science, ročník 2012, č. 7119, 2012: s. 123–131, ISSN 0302-9743.
URL <http://www.fit.vutbr.cz/research/viewpub.php.cs?id=9725>
- [13] Lamport, L.: *Time, clocks, and the ordering of events in a distributed system*. Communications of the ACM, ročník 21, 1978: s. 558–565, doi:10.1145/359545.359563.
URL <http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf>
- [14] Mehner, K.; Wagner, A.: *Visualizing the synchronization of Java-threads with UML*. In Proceeding 2000 IEEE International Symposium on Visual Languages, IEEE, 2000, s. 199–206, doi:10.1109/VL.2000.874384.
URL <http://ieeexplore.ieee.org.ezproxy.lib.vutbr.cz/document/874384/>
- [15] Meyer, B.: *Applying Design by Contract*. In Computer, IEEE Computer Society Press Los Alamitos, CA, USA, 1992, s. 40–51, doi:10.1109/2.161279.
URL <http://se.ethz.ch/~meyer/publications/computer/contract.pdf>
- [16] Oracle: *Java documentation*. [Online; navštíveno 08.01.2017].
URL <https://docs.oracle.com/javase/8/docs/>
- [17] Oracle: *Processes and Threads*. [Online; navštíveno 08.01.2017].
URL <https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html>
- [18] Per, B. H.: *Operating system principles*. Englewood Cliffs, 2003, ISBN 0-13-026611-6.
- [19] Pozniansky, E.; Schuster, A.: *MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs: Research Articles*. Concurrency and Computation: Practice & Experience - Parallel and Distributed Systems: Testing and Debugging (PADTAD), ročník 19, č. 3, 2007: s. 327–340, doi:10.1002/cpe.v19:3.
URL <http://dl.acm.org/citation.cfm?id=1228969>
- [20] Savage, S.; Burrows, M.; Nelson, G.; aj.: *Eraser: A Dynamic Data Race Detector for Multi-threaded Programs*. In Proceedings of the sixteenth ACM symposium on Operating systems principles, ACM Transactions on Computer Systems, 1997.
URL <http://homes.cs.washington.edu/~tom/pubs/eraser.pdf>
- [21] Stallings, W.: *Operating Systems: Internals and Design Principles*. Pearson, 8 vydání, 2015, ISBN 978-0133805918.
- [22] Stevens, R.: *UNIX Network Programming Second Edition: Interprocess Communications*, ročník 2. Prentice Hall, 1999, ISBN 0-13-081081-9.

Seznam příloh

Příloha A

Návod na instalaci a spuštění

Tato příloha popisuje instalaci a první spuštění nástroje RoadRunner a platformy SearchBestie.

A.1 Instalace Roadrunneru

1. Získání zdrojových souborů:

Na DVD se soubory nacházejí v adresáři:

```
roadrunner/
```

Z Git repozitáře:

```
git clone git@pajda.fit.vutbr.cz:jct/roadrunner.git
git checkout contract-validator
```

2. Sestavení projektu a nastavení systémových proměnných:

```
cd roadrunner
```

```
ant
source msetup
```

3. Základní spuštění:

```
javac test/*
rrrun -tool= test.Test
```

4. Příklad spuštění kontrakt analyzátoru:

```
javac .test/src/cz/vutbr/fit/*
rrrun -classpath=.test/src -tool=CT
-contractFile=.test/contract-config/test1 cz.vutbr.fit.Test1
```

A.2 Instalace SearchBestie

1. Přednastavení prostředí:

Je nutné mít nastavenou proměnnou \$JAVA_HOME obsahující cestu k JDK ve verzi 1.6 nebo vyšší.

2. Získání zdrojových souborů:

Na DVD se soubory nacházejí v adresáři:

```
searchbestie/
```

Z Git repozitáře:

```
git clone git@pajda.fit.vutbr.cz:jct/searchBestie.git  
git checkout contract-validator-integration
```

3. Sestavení projektu:

```
cd searchBestie  
gradle fatJar
```

4. Základní spuštění:

Musí být zadán konfigurační soubor (v příkladu je ./tmp_experiments/tmp_contract.xml), ve kterém je třeba upravit tyto dvě proměnné:

- (3. řádek) **rr-home-path** – udává cestu k Roadrunneru.
- (4. řádek) **output-path** – udává cestu ke složce, kde se budou ukládat výstupní soubory Roadrunneru.

```
java -jar build/libs/searchBestie-all.jar -config  
./tmp_experiments/tmp_contract.xml
```

Příloha B

Testy funkčnosti dynamického analyzátoru

Tato kapitola obsahuje popis 46 testů, které byly vytvořeny na ověření funkčnosti. Nejdříve jsou popsány testy zaměřené na jednoduché kontrakty, následně na použití vektorového času, dále na složitější kontrakty. Závěrem jsou popsány testy zaměřené na kontrakty s parametry a návratovou hodnotu.

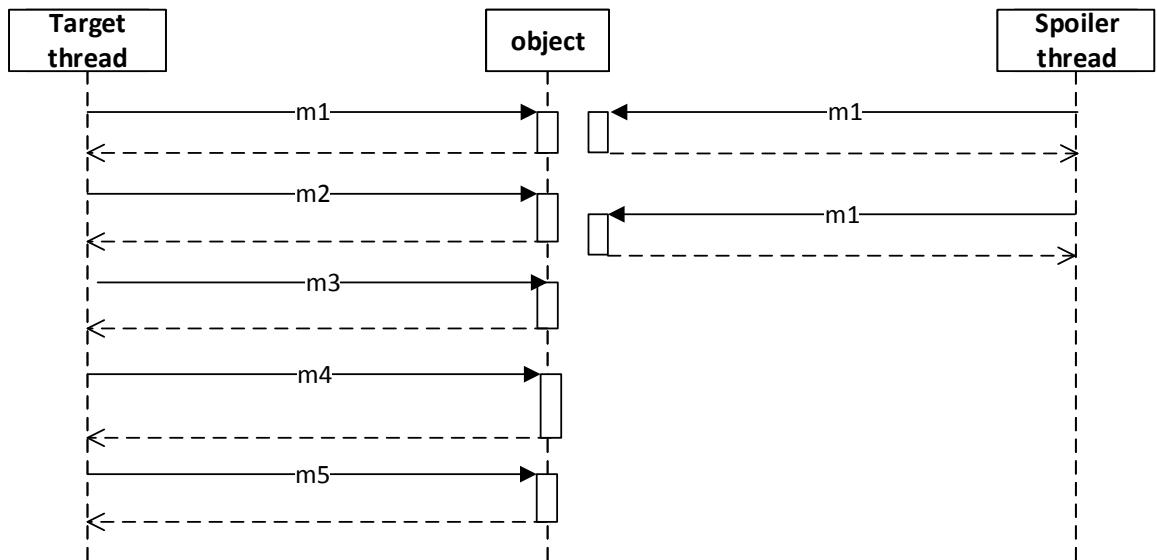
B.1 Testování jednoduchých kontraktů

Test 1

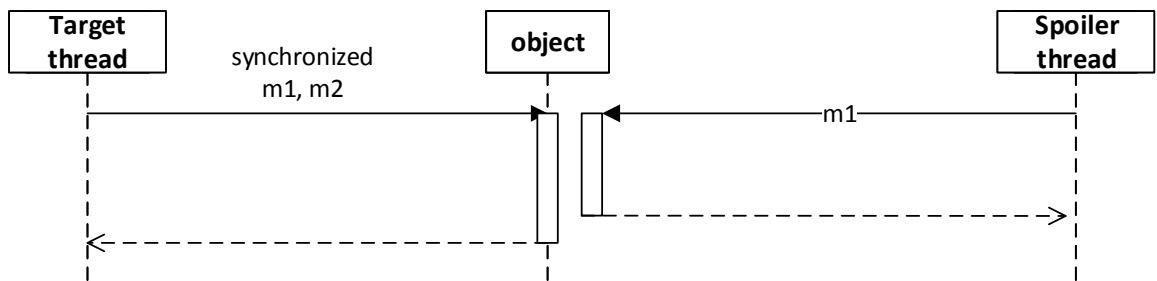
- Kategorie: JEDNODUCHÝ KONTRAKT.
- Popis: Obě vlákna využívají korektní synchronizaci.
- Kontrakt: $m1() \ m2() <- \{ m1() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**

Test 2

- Kategorie: JEDNODUCHÝ KONTRAKT.
- Popis: Vlákno *Spoiler thread* volá metodu *m1* bez synchronizace.
- Kontrakt: $m1() \ m2() <- \{ m1() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



Obrázek B.1: Diagram testu 1.



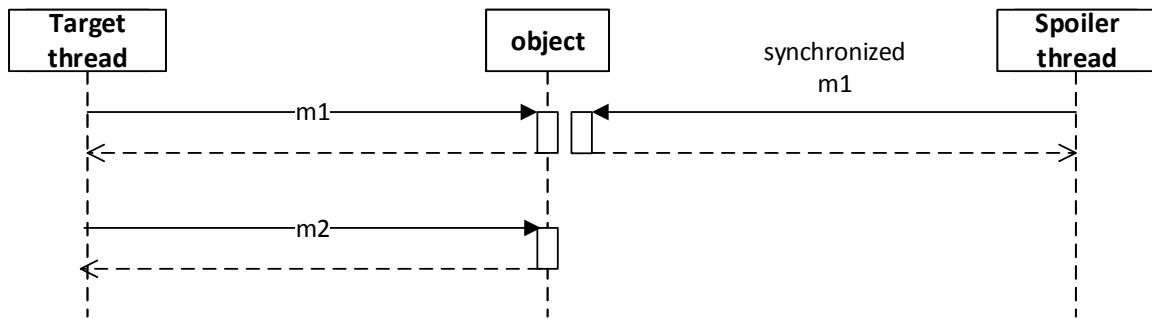
Obrázek B.2: Diagram testu 2.

Test 3

- Kategorie: JEDNODUCHÝ KONTRAKT.
- Popis: Vlákno *Target thread* volá metody *m1* a *m2* bez synchronizace.
- Kontrakt: $m1() \quad m2() \leftarrow \{ m1() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**

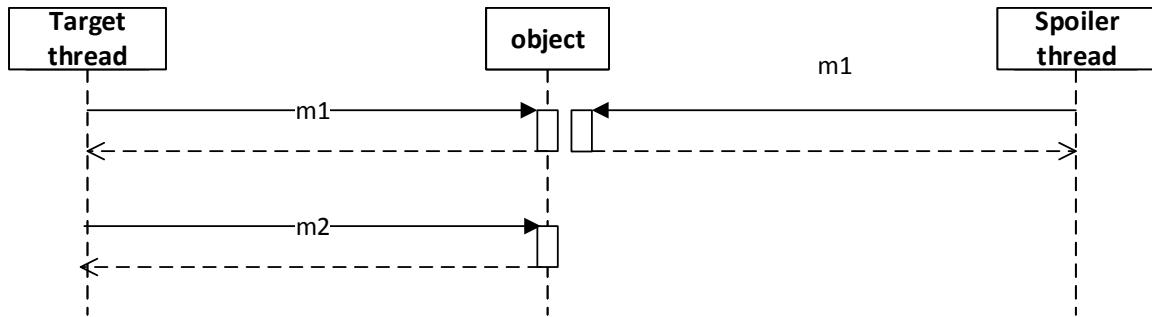
Test 4

- Kategorie: JEDNODUCHÝ KONTRAKT.
- Popis: Obě vlákna volají všechny metody bez synchronizace.
- Kontrakt: $m1() \quad m2() \leftarrow \{ m1() \}$



Obrázek B.3: Diagram testu 3.

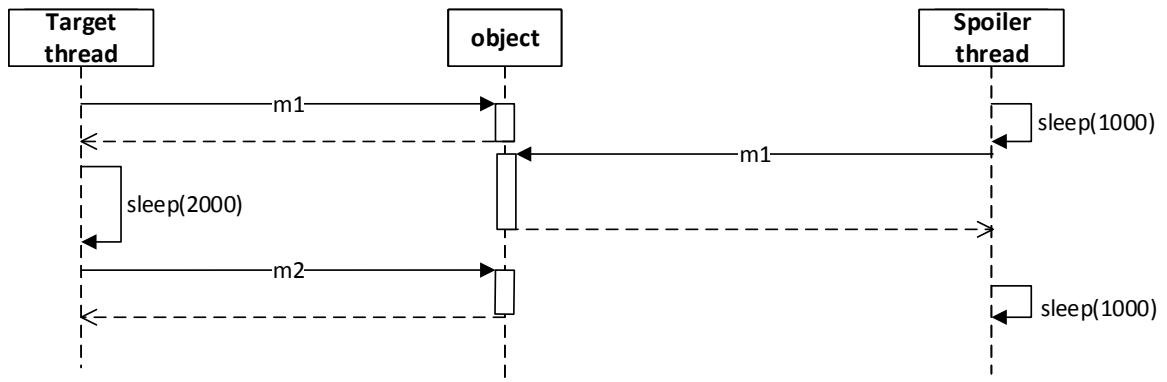
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



Obrázek B.4: Diagram testu 4.

Test 5

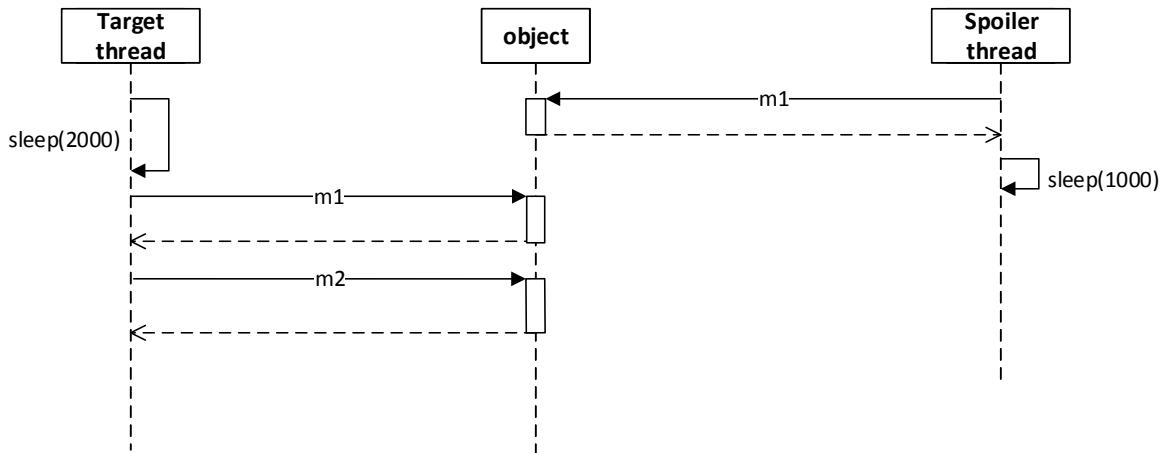
- Kategorie: JEDNODUCHÝ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace. Pomocí uspání vláken, je spojler v čase vykonán mezi první a poslední metodou targetu. K detekování není nutný vektorový čas.
- Kontrakt: `m1() m2() <- { m1() }`
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



Obrázek B.5: Diagram testu 5.

Test 6

- Kategorie: JEDNODUCHÝ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace. Pomocí uspání vláken, je spoiler v čase vykonán dříve, než je vykonána první metoda targetu. K detekování je nutný vektorový čas.
- Kontrakt: $m1() \ m2() \leftarrow \{ m1() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**

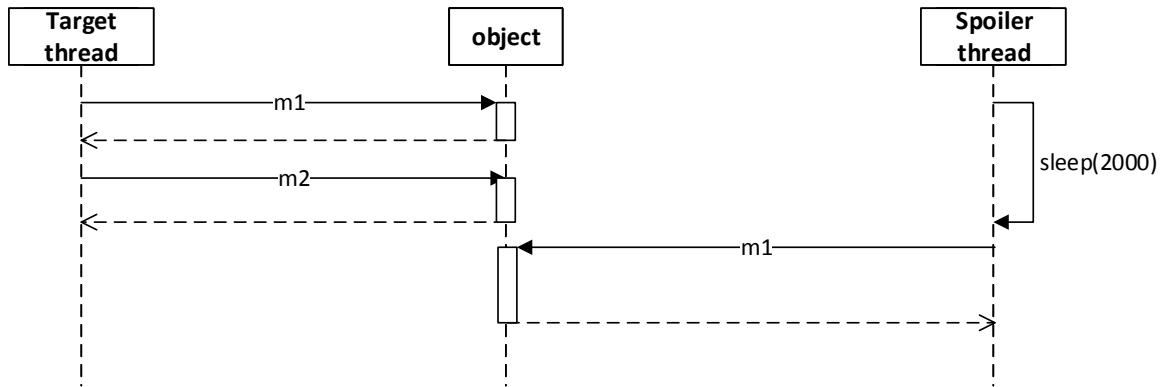


Obrázek B.6: Diagram testu 6.

Test 7

- Kategorie: JEDNODUCHÝ KONTRAKT, VEKTOROVÝ ČAS.

- Popis: Obě vlákna volají všechny metody bez synchronizace. Pomocí uspání vláken, je celý target v čase vykonán dříve, než první metoda spoileru. K detekování je nutný vektorový čas.
- Kontrakt: $m1() \ m2() \leftarrow \{ m1() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



Obrázek B.7: Diagram testu 7.

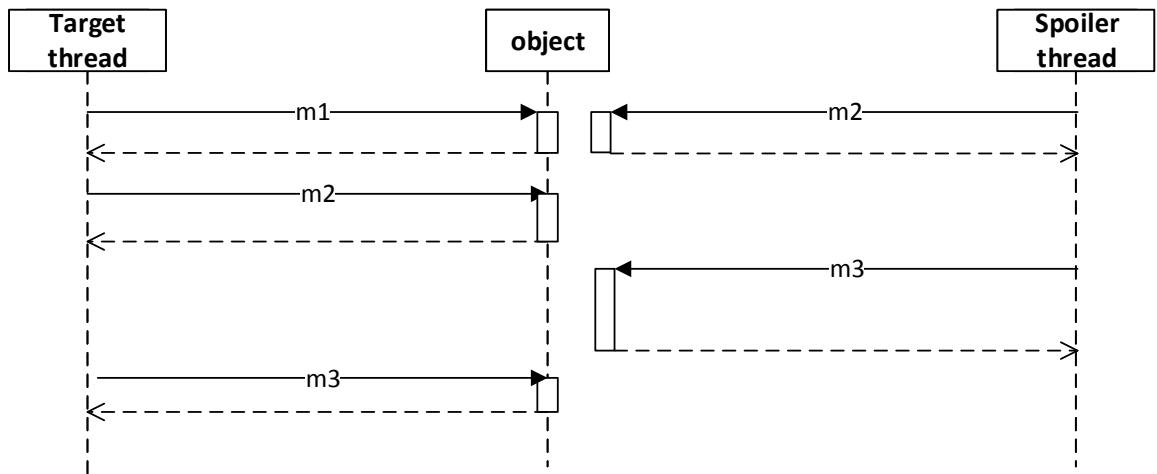
B.2 Testování složitějších kontraktů

Test 8

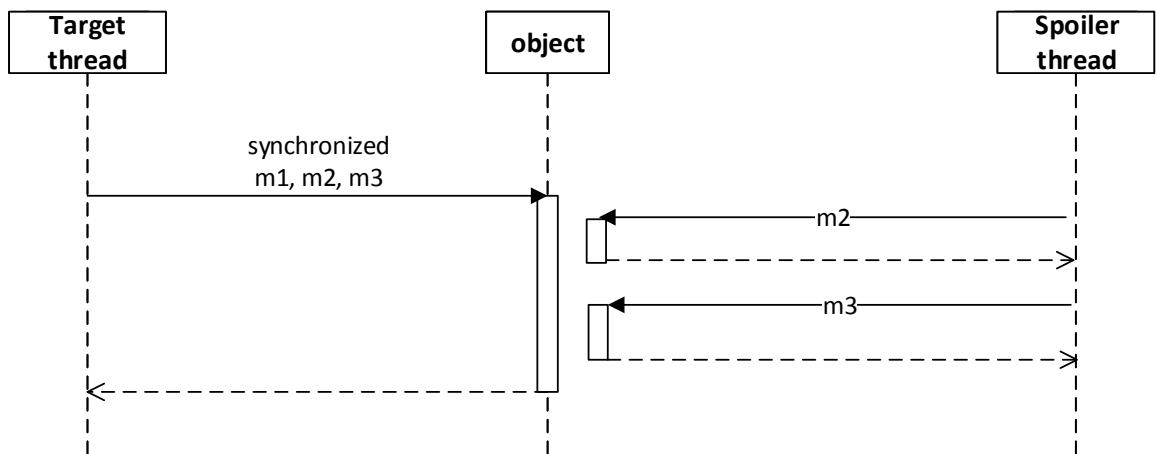
- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m2() \ m3() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**

Test 9

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno *Spoiler thread* volá všechny metody bez synchronizace.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m2() \ m3() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



Obrázek B.8: Diagram testu 8.



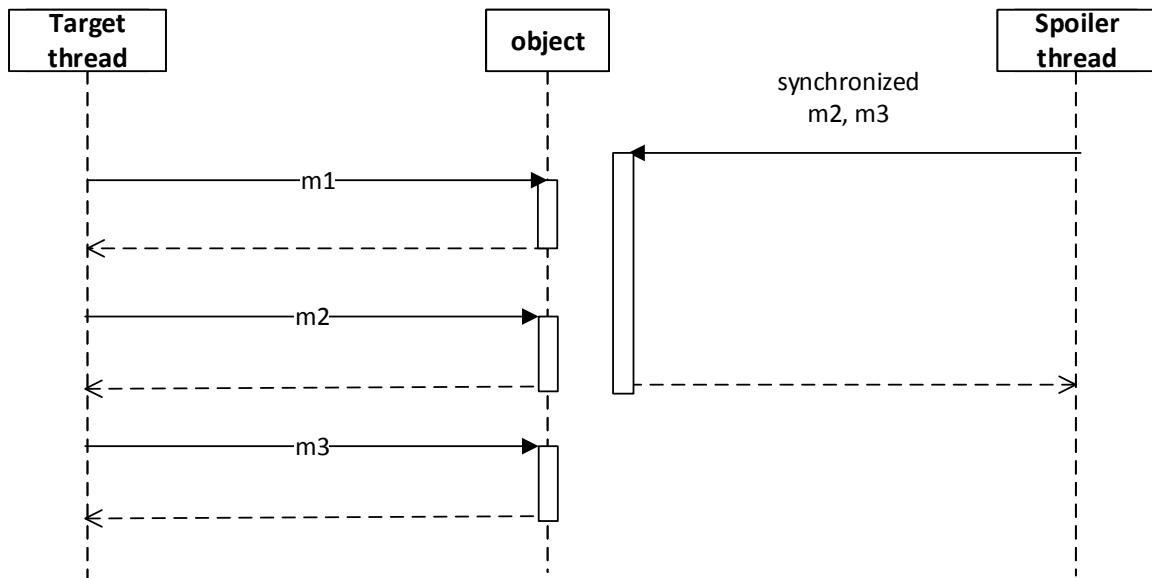
Obrázek B.9: Diagram testu 9.

Test 10

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno *Target thread* volá všechny metody bez synchronizace.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m2() \ m3() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**

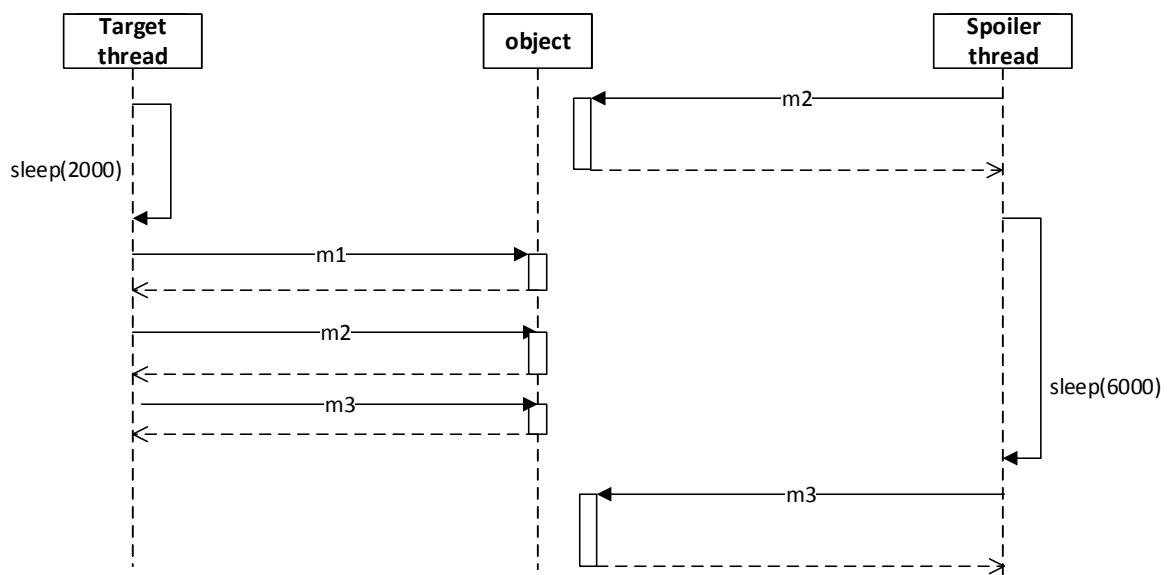
Test 11

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.



Obrázek B.10: Diagram testu 10.

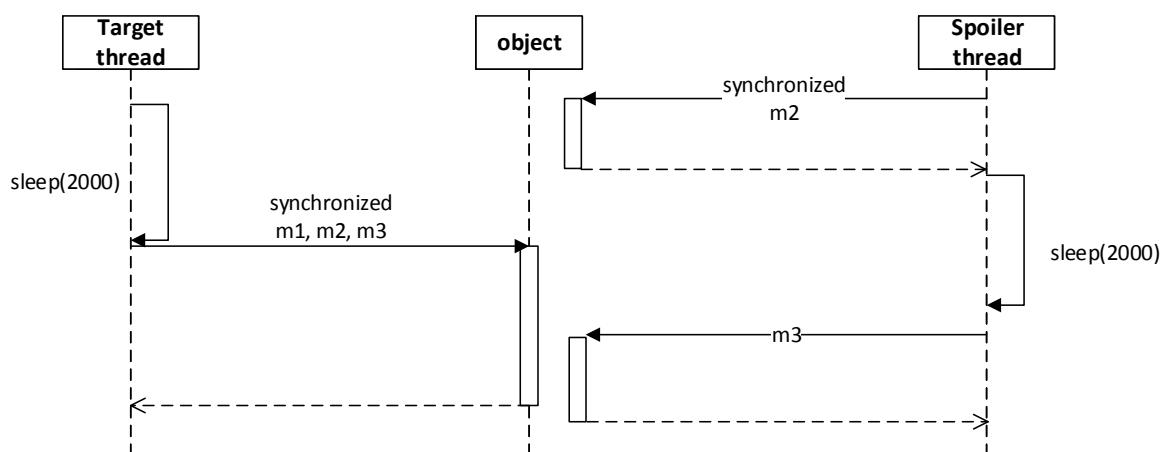
- Popis: Obě vlákna volají všechny metody bez synchronizace. Pomocí uspání vláken, je target v čase vykonán mezi první a poslední metodou spoileru. K detekování je nutný vektorový čas.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m2() \ m3() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- Test proběhl úspěšně.**



Obrázek B.11: Diagram testu 11.

Test 12

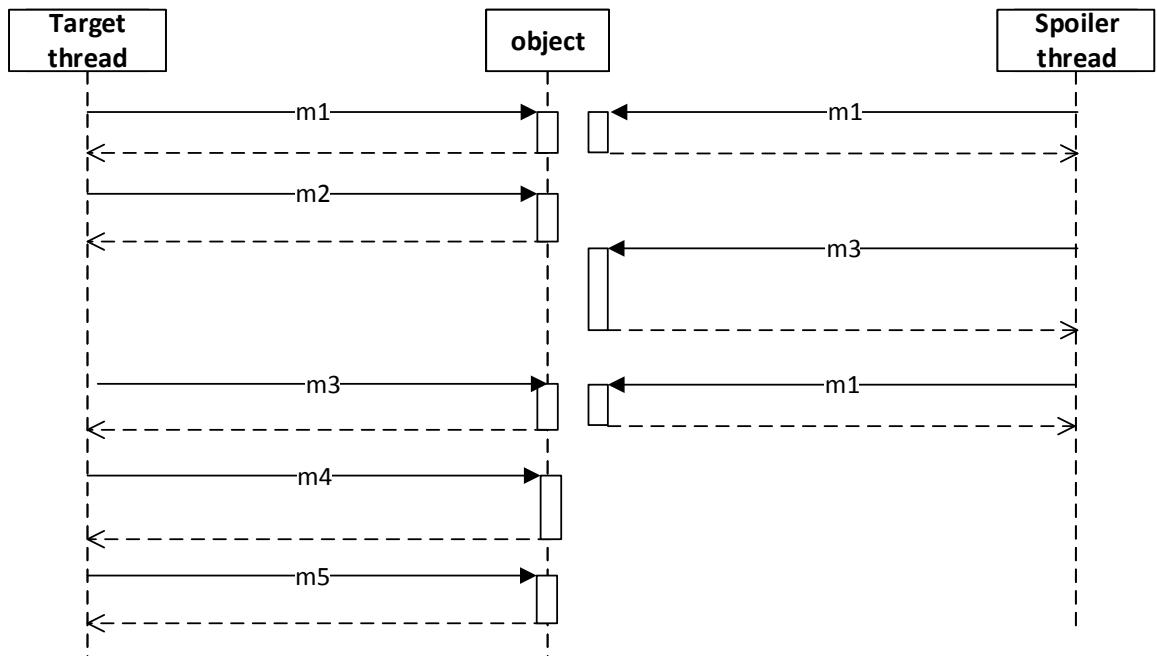
- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno *Target thread* volá všechny metody za použití správné synchronizace. Metoda *m2* vlákna *Spoiler thread* volá první metodu za použití synchronizace, ale druhou nikoli. K porušení konaktu nedochází, neboť spoiler nemůže být celý vložen v targetu.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m2() \ m3() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



Obrázek B.12: Diagram testu 12.

Test 13

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace.
- Kontrakt: $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow \{ m1() \ m3() \ m1() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



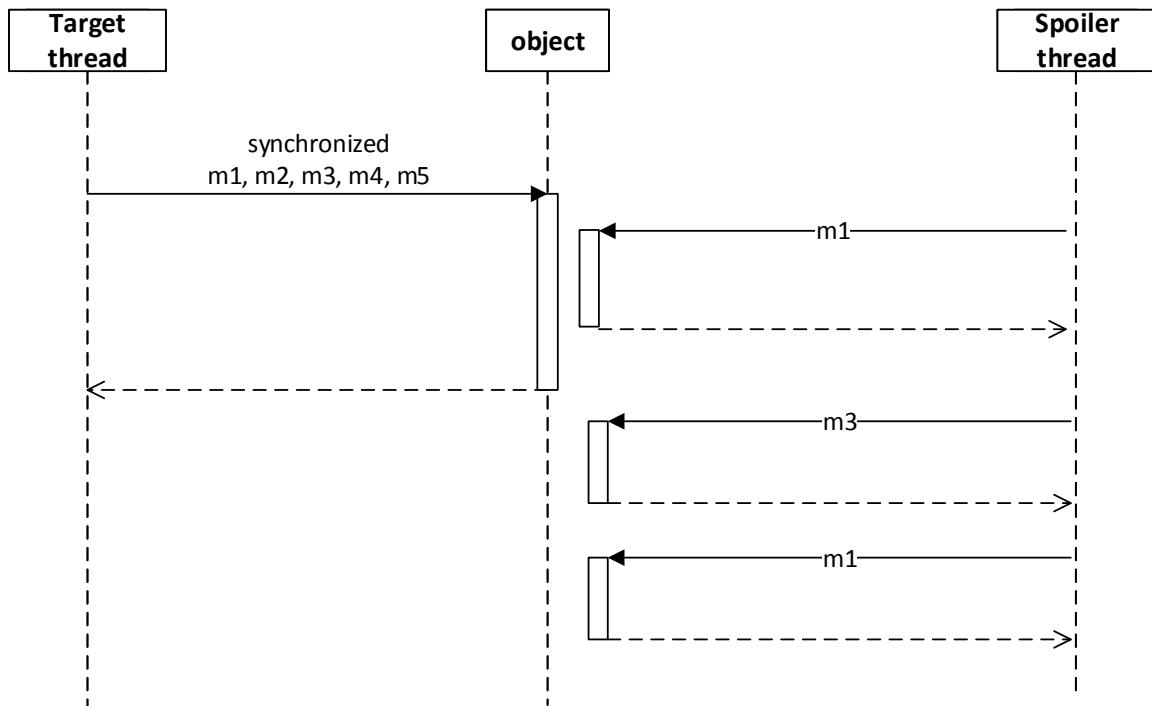
Obrázek B.13: Diagram testu 13.

Test 14

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno *Spoiler thread* volá všechny metody bez synchronizace.
- Kontrakt: $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow \{ m1() \ m3() \ m1() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**

Test 15

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno *Target thread* volá všechny metody bez synchronizace.
- Kontrakt: $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow \{ m1() \ m3() \ m1() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



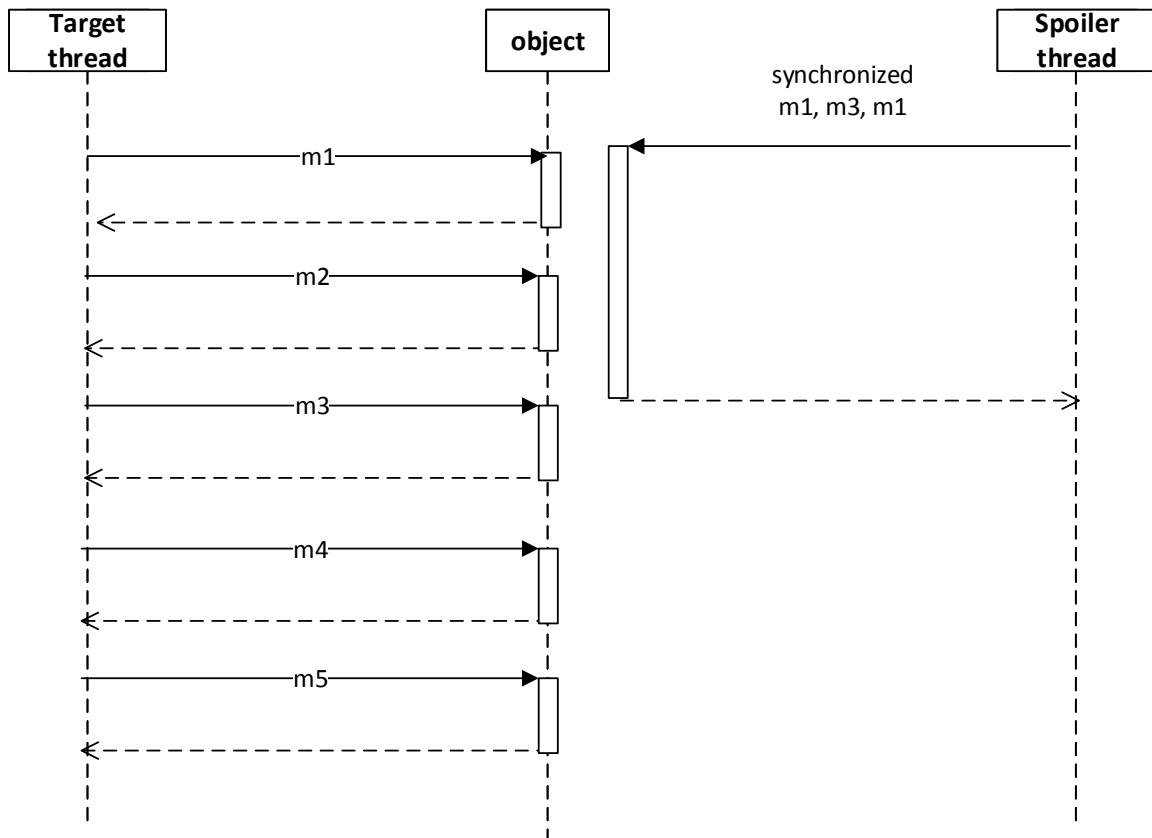
Obrázek B.14: Diagram testu 14.

Test 16

- Kategorie: IGNOROVÁNÍ OSTATNÍCH METOD.
- Popis: Ve vlákně *Target thread* i *Spoiler thread* vznikne více instancí targetů a spoilerů. Testuje se ignorování ostatních metod.
- Kontrakt: $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow \{ m1() \ m3() \ m1() \}$
- Očekávaný výsledek: 4 nalezených porušení.
- Skutečný výsledek: 4 nalezených porušení.
- **Test proběhl úspěšně.**

Test 17

- Kategorie: NEDOKONČENÉ SEKVENCE.
- Popis: Target ve vlákně *Target thread* není dokončen - chybí poslední metoda.
- Kontrakt: $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow \{ m1() \ m3() \ m1() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



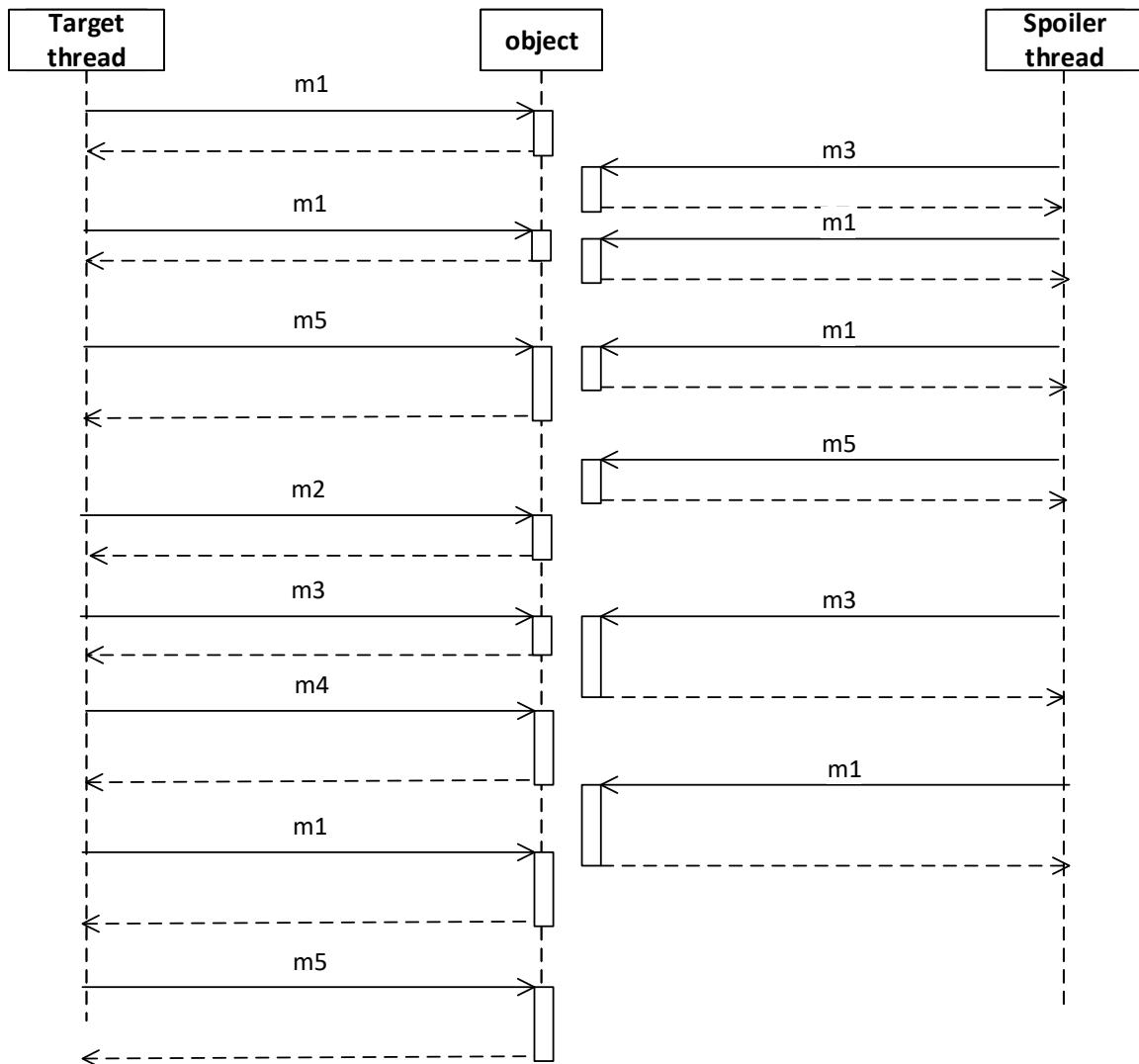
Obrázek B.15: Diagram testu 15.

Test 18

- Kategorie: NEDOKONČENÉ SEKVENCE.
- Popis: Spoiler ve vlákně *Spoiler thread* není dokončen - chybí první metoda.
- Kontrakt: $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow \{ m1() \ m3() \ m1() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**

Test 19

- Kategorie: NEDOKONČENÉ SEKVENCE.
- Popis: Spoiler ve vlákně *Spoiler thread* není dokončen - chybí poslední metoda.
- Kontrakt: $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow \{ m1() \ m3() \ m1() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.

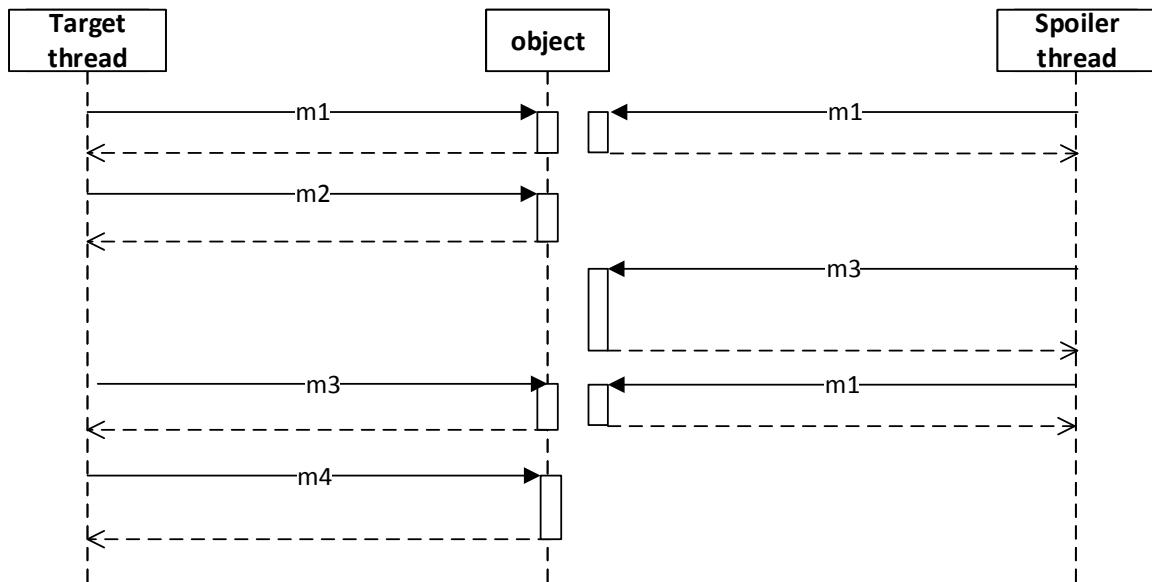


Obrázek B.16: Diagram testu 16.

- **Test proběhl úspěšně.**

Test 20

- Kategorie: NEDOKONČENÉ SEKVENCE.
- Popis: Spoiler ve vlákně *Spoiler thread* není dokončen - chybí prostřední metoda.
- Kontrakt: $m1() \circ m2() \circ m3() \circ m4() \circ m5() \leftarrow \{ m1() \circ m3() \circ m1() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



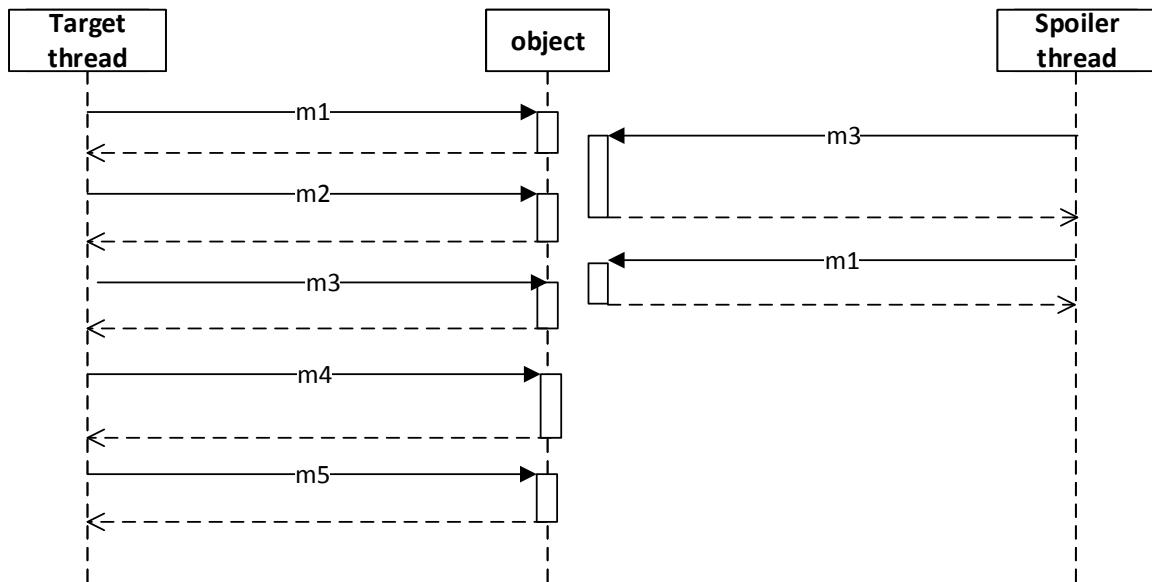
Obrázek B.17: Diagram testu 17.

Test 21

- Kategorie: VÍCE OBJEKTŮ.
- Popis: Target ve vlákně *Target thread* je volán na objekt *object1*, ale spoiler ve vlákně *Spoiler thread* je volán na objekt *object2*.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m2() \ m3() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**

Test 22

- Kategorie: VÍCE OBJEKTŮ.
- Popis: Spoiler ve vlákně *Spoiler thread* volá první metodu *m2* na objekt *object1*, ale druhou metodu *m3* na objekt *object2*.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m2() \ m3() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



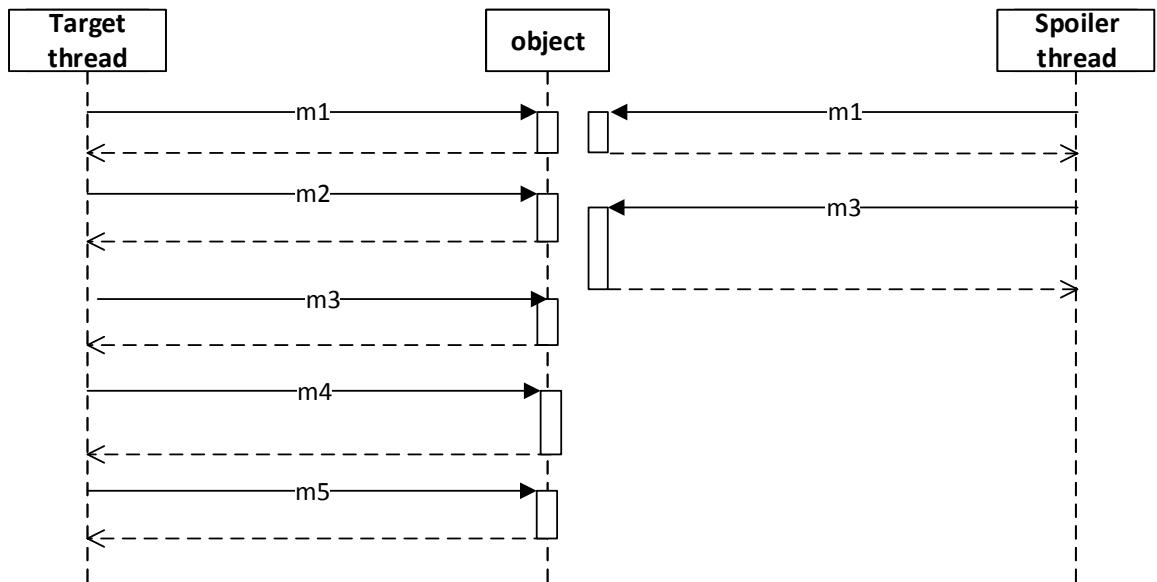
Obrázek B.18: Diagram testu 18.

Test 23

- Kategorie: VÍCE OBJEKTŮ.
- Popis: Target ve vlákně *Target thread* volá první metodu *m2* a poslední metodu *m3* na objekt *object1*, ale druhou metodu *m2* na objekt *object2*.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m2() \ m3() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**

Test 24

- Kategorie: VÍCE SPOILERŮ.
- Popis: Je provedena pouze jedna definice spoileru. Dojde k nalezení 2 porušení, neboť spoiler i target mohou být nalezeny jako ve vlákně *Target thread*, tak i *Spoiler thread*.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m1() \ m3() \mid m2() \ m4() \mid m5() \}$
- Očekávaný výsledek: 2 nalezených porušení.
- Skutečný výsledek: 2 nalezených porušení.
- **Test proběhl úspěšně.**



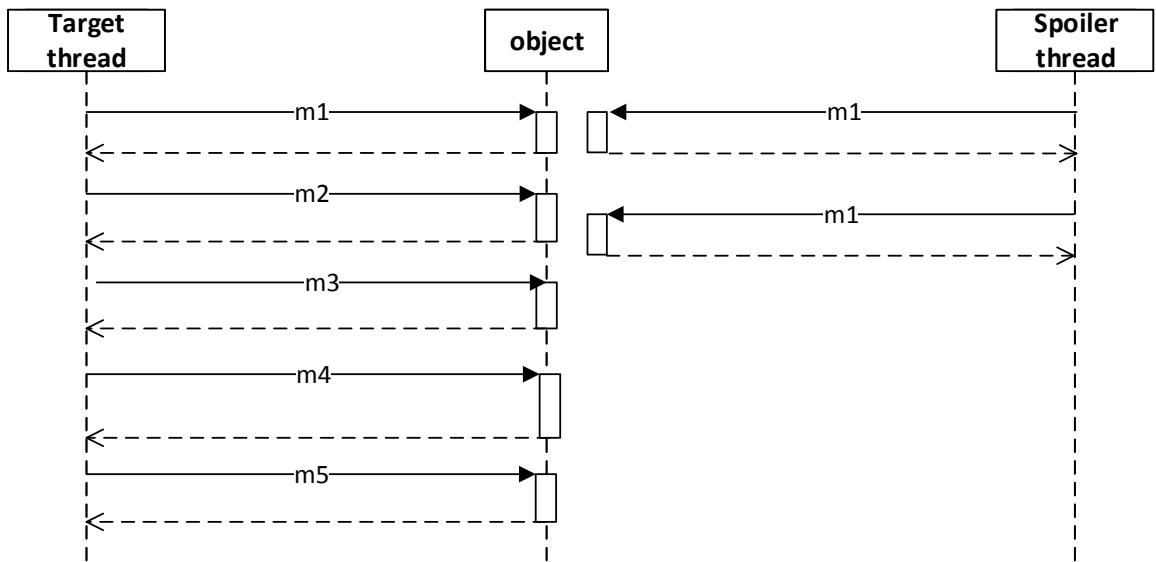
Obrázek B.19: Diagram testu 19.

Test 25

- Kategorie: VÍCE SPOILERŮ.
- Popis: Dojde k nalezení 2 různých spoilerů ($m1()$ $m3()$ a $m5()$) ve vlákně *Spoiler thread*.
- Kontrakt: $m1() \quad m2() \quad m3() \leftarrow \{ \quad m1() \quad m3() \mid m2() \quad m4() \mid m5() \}$
- Očekávaný výsledek: 3 nalezených porušení.
- Skutečný výsledek: 3 nalezených porušení.
- **Test proběhl úspěšně.**

Test 26

- Kategorie: VÍCE SPOILERŮ, VÍCE VLÁKEN.
- Popis: Dojde k nalezení 2 různých spoilerů ($m1()$ $m3()$ a $m5()$) ve vláknech *Spoiler thread1* a *Spoiler thread2*.
- Kontrakt: $m1() \quad m2() \quad m3() \leftarrow \{ \quad m1() \quad m3() \mid m2() \quad m4() \mid m5() \}$
- Očekávaný výsledek: 2 nalezených porušení.
- Skutečný výsledek: 2 nalezených porušení.
- **Test proběhl úspěšně.**



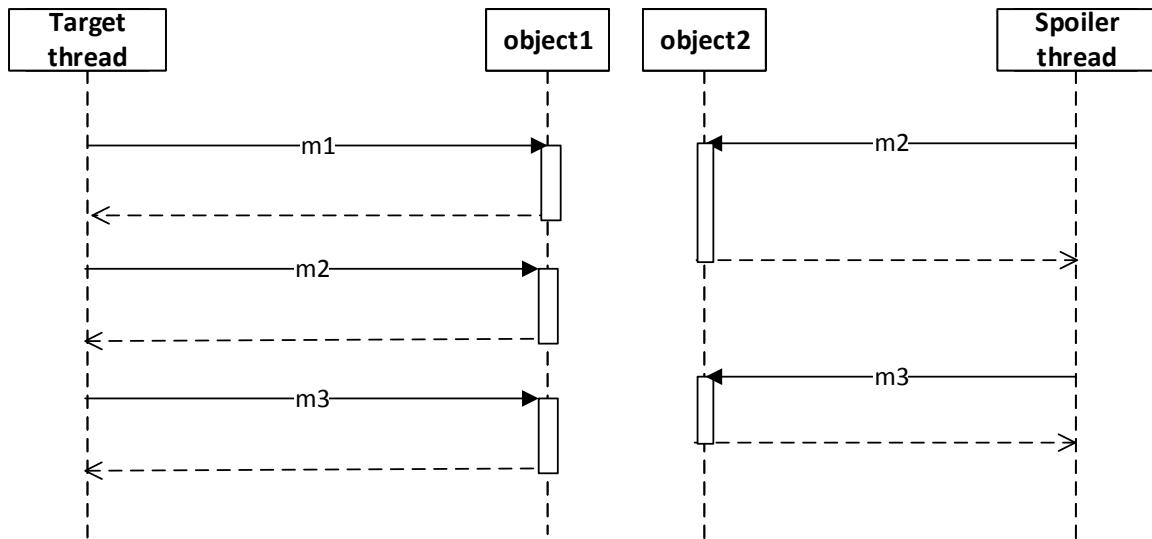
Obrázek B.20: Diagram testu 20.

Test 27

- Kategorie: VÍCE SPOILERŮ, VÍCE VLÁKEN.
- Popis: Nedojde k nalezení žádného spoileru, neboť spoiler musí být vykonán v jednom vlákně.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m1() \ m3() \mid m2() \ m4() \mid m5() \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**

Test 28

- Kategorie: VÍCE SPOILERŮ, VÍCE VLÁKEN.
- Popis: Dojde k nalezení pouze jednoho spoileru ($m2() \ m4()$) ve vlákně *Spoiler thread3*, neboť v ostatních vláknech je použita správná synchronizace.
- Kontrakt: $m1() \ m2() \ m3() \leftarrow \{ m1() \ m3() \mid m2() \ m4() \mid m5() \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



Obrázek B.21: Diagram testu 21.

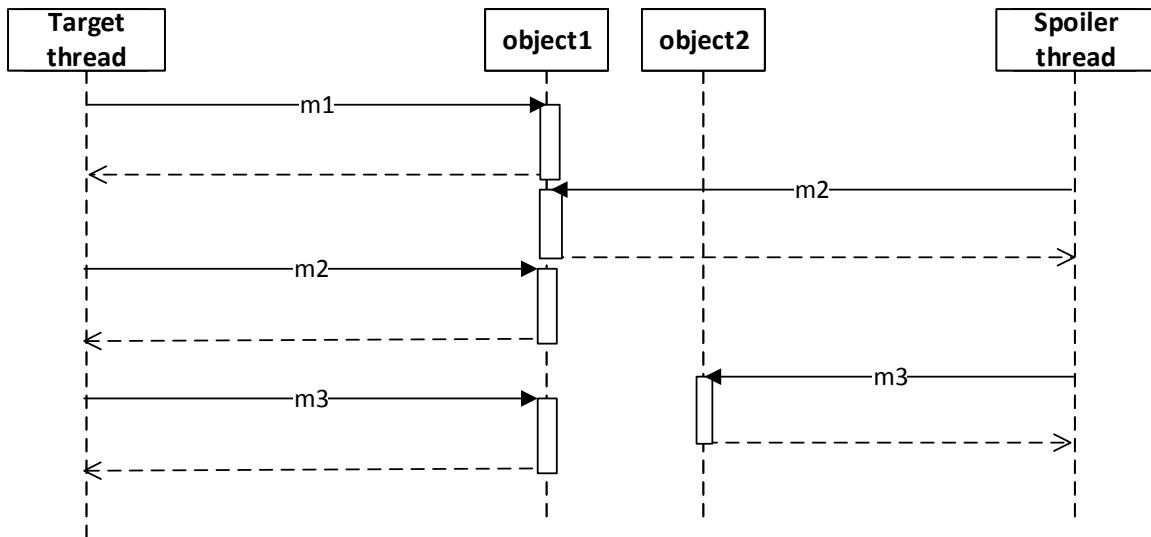
B.3 Testování kontraktů s parametry

Test 29

- Kategorie: PARAMETRY.
- Popis: Parametry metod se shodují s definicí konaktu.
- Kontrakt: $m1(A) \ m2(B) \leftarrow \{ m3(C) \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**

Test 30

- Kategorie: PARAMETRY.
- Popis: Metoda $m2()$ je zavolána bez parametru.
- Kontrakt: $m1(A) \ m2(B) \leftarrow \{ m3(C) \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



Obrázek B.22: Diagram testu 22.

Test 31

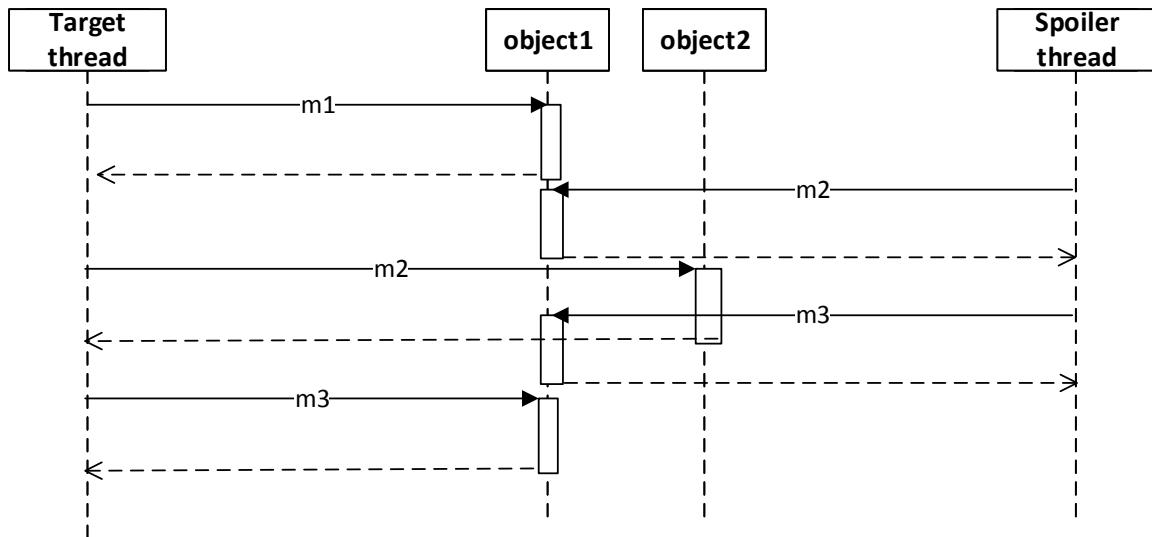
- Kategorie: PARAMETRY.
- Popis: Parametry metod $m1$ a $m3$ se shodují.
- Kontrakt: $m1(A) \ m2(B) \leftarrow \{ \ m3(A) \ }$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 32

- Kategorie: PARAMETRY.
- Popis: Parametry metod $m1$ a $m3$ se neshodují.
- Kontrakt: $m1(A) \ m2(B) \leftarrow \{ \ m3(A) \ }$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 33

- Kategorie: RŮZNÉ TYPY PARAMETRŮ.
- Popis: Parametry v metodách $m1$ a $m3$ jsou typu *String*, oproti tomu parametr metody $m2(2)$ je primitivního typu *int*.
- Kontrakt: $m1(A) \ m2(B) \leftarrow \{ \ m3(A) \ }$



Obrázek B.23: Diagram testu 23.

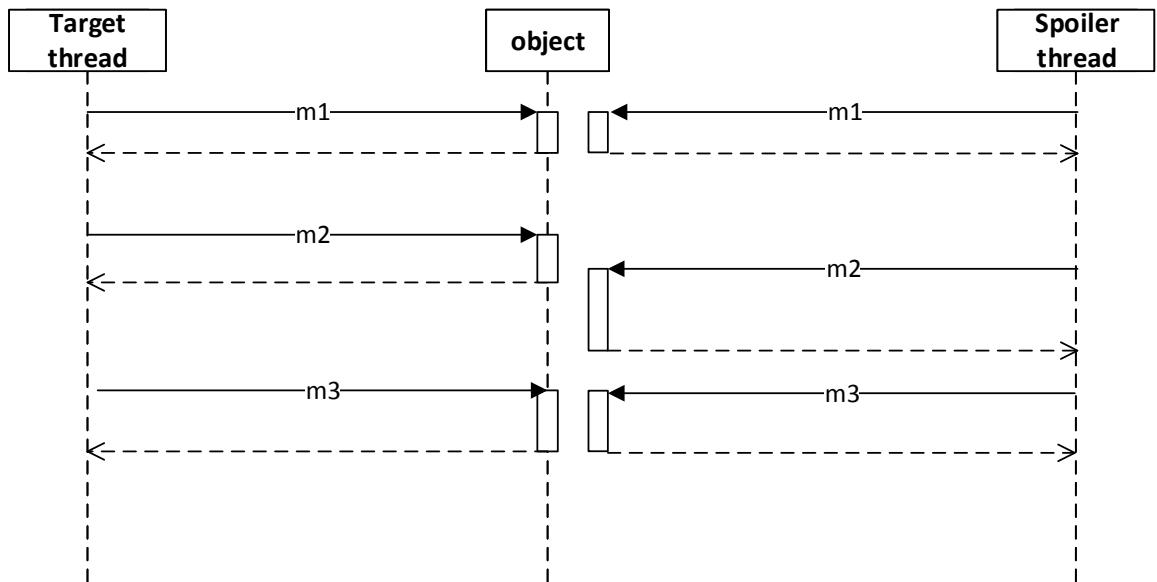
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 34

- Kategorie: RŮZNÉ TYPY PARAMETRŮ.
- Popis: Parametry v metodách $m1$ a $m3$ jsou typu *Double*, oproti tomu parametr metody $m2$ je primitivního typu *int*.
- Kontrakt: $m1(A) \ m2(B) \leftarrow \{ \ m3(A) \ }$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 35

- Kategorie: RŮZNÉ TYPY PARAMETRŮ.
- Popis: Parametry v metodách $m1$ a $m3$ jsou instance nové třídy *MyObject* s definovanou *equals* metodou, oproti tomu parametr metody $m2$ je primitivního typu *int*.
- Kontrakt: $m1(A) \ m2(B) \leftarrow \{ \ m3(A) \ }$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



Obrázek B.24: Diagram testu 24.

Test 36

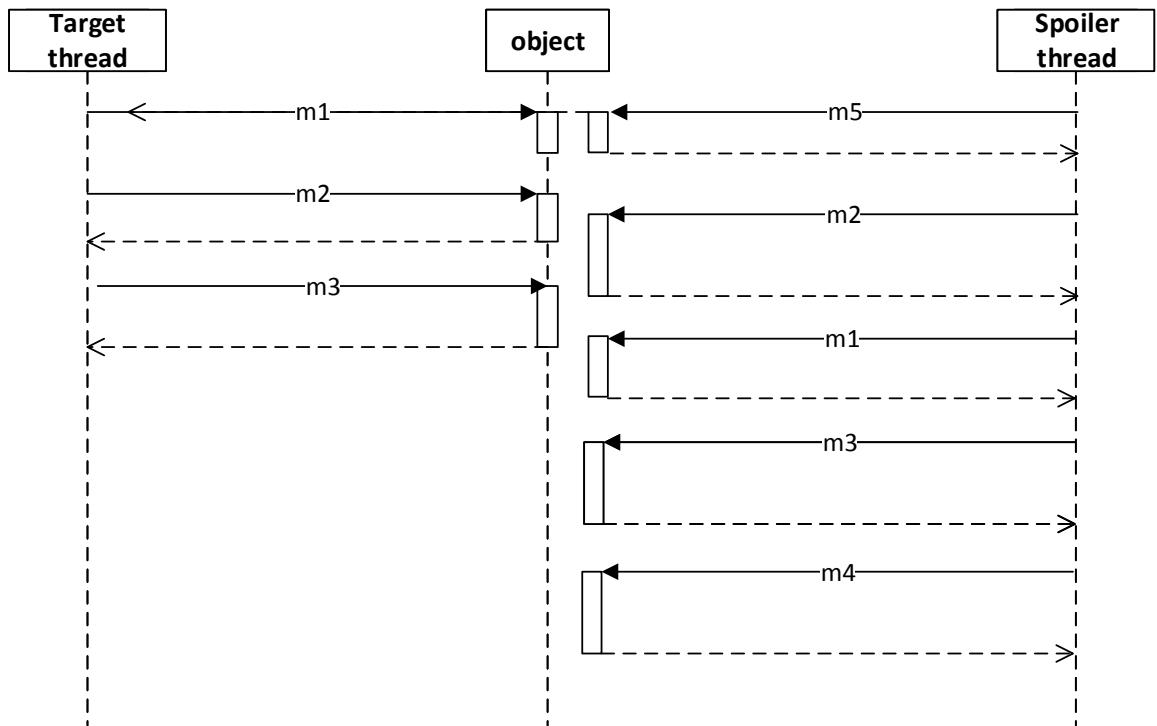
- Kategorie: IGNOROVÁNÍ PARAMETRŮ.
- Popis: Parametr metod $m1$ a $m3$ je ignorován, takže nezáleží na jeho hodnotě.
- Kontrakt: $m1(_) \ m2(X) \leftarrow \{ \ m3(_) \ }$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 37

- Kategorie: IGNOROVÁNÍ PARAMETRŮ.
- Popis: Parametr metody $m1$ není zadán. Hodnota a typ parametru jsou ignorovány, ale počet musí souhlasit. Tím pádem není nalezena žádná instance targetu.
- Kontrakt: $m1(_) \ m2(X) \leftarrow \{ \ m3(_) \ }$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.

Test 38

- Kategorie: VÍCE PARAMETRŮ.
- Popis: Parametry mezi targetem a spoilerem souhlasí, takže porušení je nalezeno.
- Kontrakt: $m1(A,B) \ m2(A,C) \leftarrow \{ \ m3(A,B,C) \ }$



Obrázek B.25: Diagram testu 25.

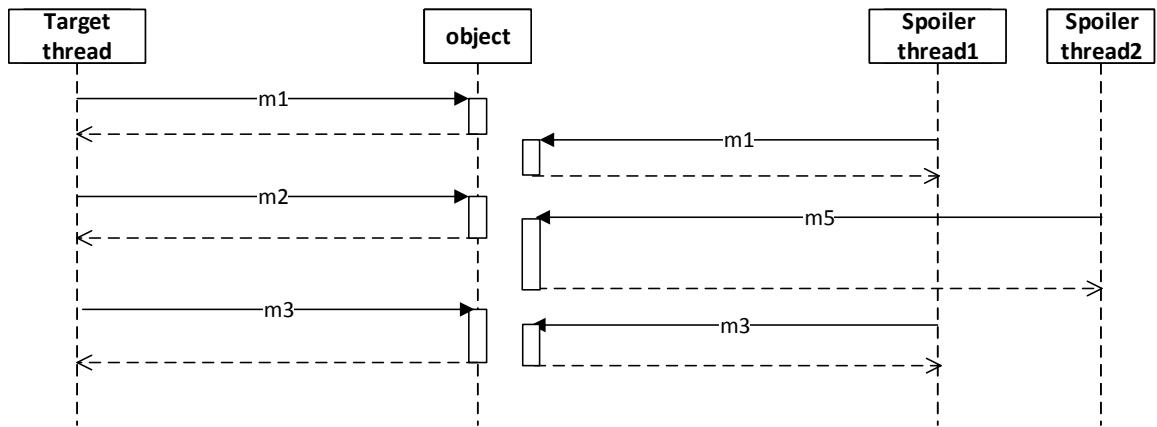
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 39

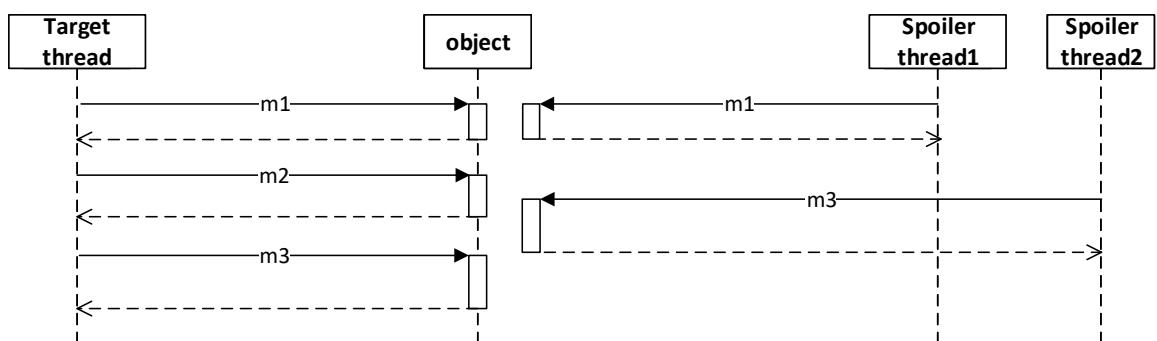
- Kategorie: VÍCE PARAMETRŮ.
- Popis: V tagetu, ve vlákně *Target thread*, se neshoduje první parametr u metod *m1* a *m2*, takže neexistuje instance targetu.
- Kontrakt: $m1(A, B) \quad m2(A, C) \leftarrow \{ \quad m3(A, B, C) \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.

Test 40

- Kategorie: VÍCE PARAMETRŮ.
- Popis: Dokončená instance tagetu obsahuje až druhou metodu *m1*, protože u první metody *m1* se neshodují parametry s metodou *m2* a ani s metodou *m3* ve vláknu spoileru *Spoiler thread*.
- Kontrakt: $m1(A, B) \quad m2(A, C) \leftarrow \{ \quad m3(A, B, C) \}$



Obrázek B.26: Diagram testu 26.



Obrázek B.27: Diagram testu 27.

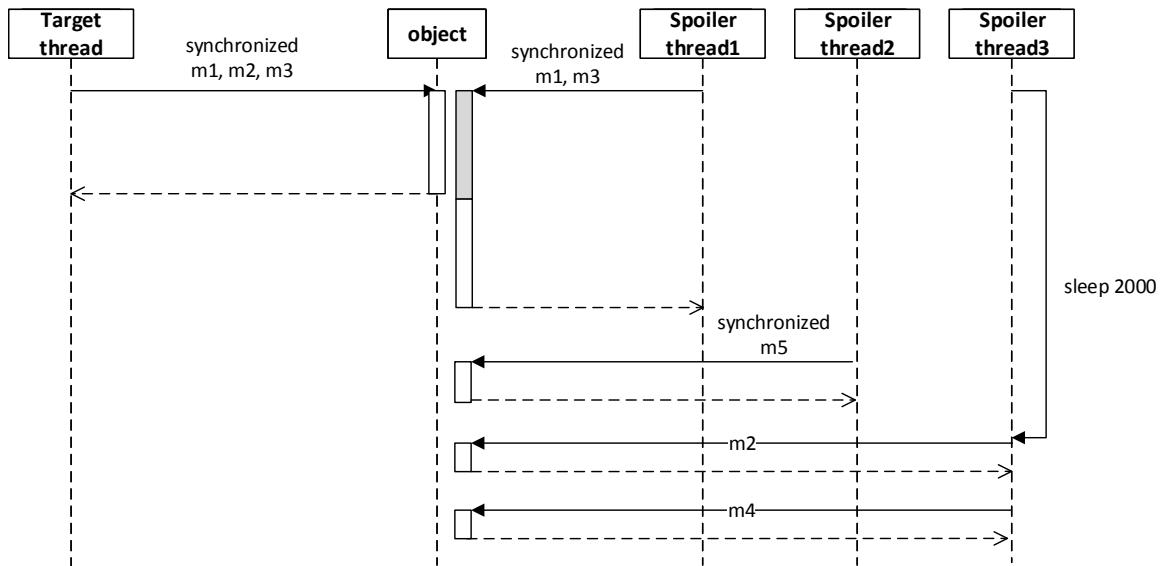
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 41

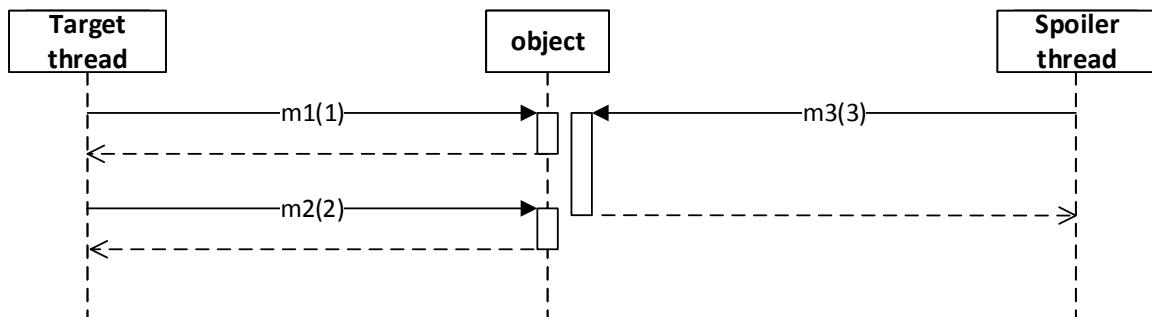
- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Návratový parametr metody *m1* je shodný s parametry metod *m2* a *m3*.
- Kontrakt: $A:m1(_) \quad m2(A) \leftarrow \{ m3(A) \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 42

- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Návratový parametr metody *m1* není shodný s parametrem metody *m3* ve vláknu spoileru *Spoiler thread*.



Obrázek B.28: Diagram testu 28.

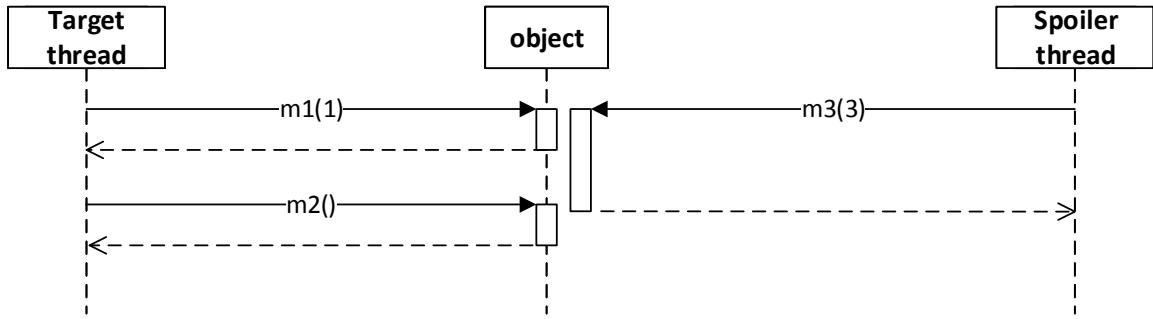


Obrázek B.29: Diagram testu 29.

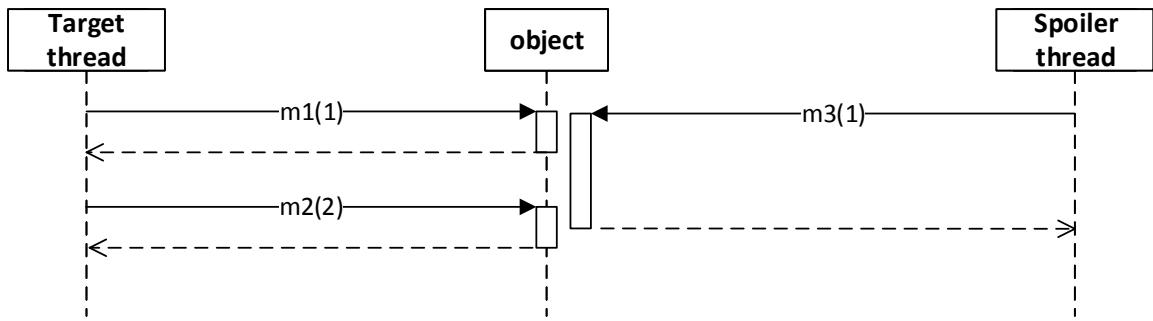
- Kontrakt: $A:m1(_) \ m2(A) \leftarrow \{ m3(A) \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.

Test 43

- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Parametry jsou shodné, ale je použita správná synchronizace a proto k porušení nedo- jde.
- Kontrakt: $A:m1(_) \ m2(A) \leftarrow \{ m3(A) \}$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.



Obrázek B.30: Diagram testu 30.



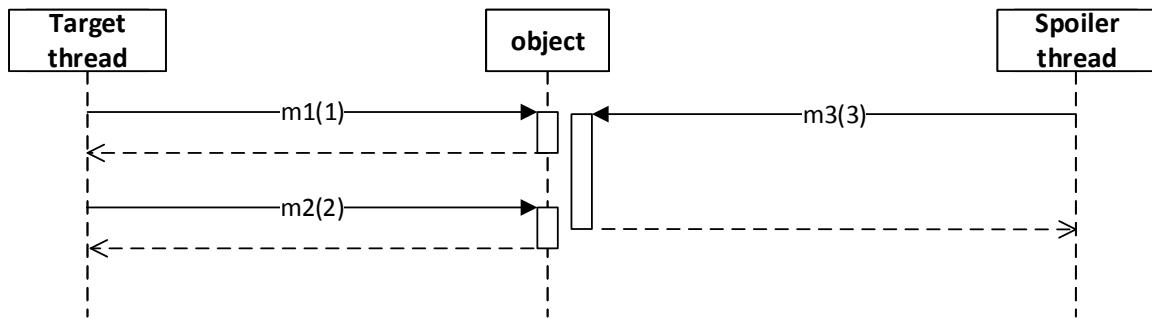
Obrázek B.31: Diagram testu 31.

Test 44

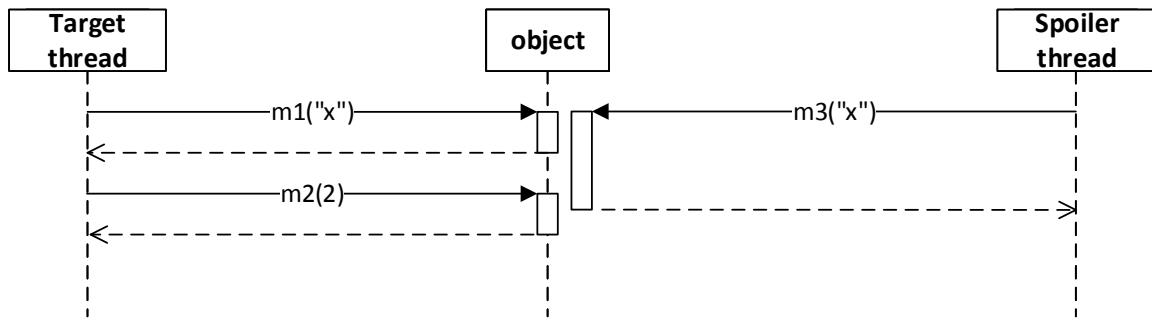
- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Návratový parametr je kontrolován u dvou metod targetu a parametry z obou metod jsou použity jako vstupní parametry do metody ve spoileru. Metody ve vláknu targetu *Target thread* jsou volány se správnou synchronizací, ale ve vláknu spoileru *Spoiler thread*, jsou volány bez synchronizace.
- Kontrakt: $X:m1() \quad Y:m2() \leftarrow \{ m3(X,Y) \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

Test 45

- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Je definováno více spoilerů na jeden target s kombinací parametrů a návratového parametru.
- Kontrakt: $X:m1(_) \quad m2(A, X) \leftarrow \{ m3(_,X) \mid m1(A) \quad X:m4(A) \}$
- Očekávaný výsledek: 2 nalezených porušení.
- Skutečný výsledek: 2 nalezených porušení.



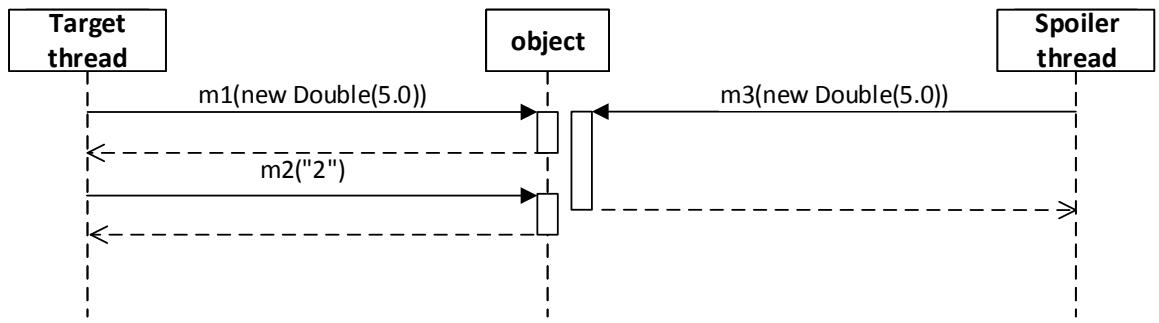
Obrázek B.32: Diagram testu 32.



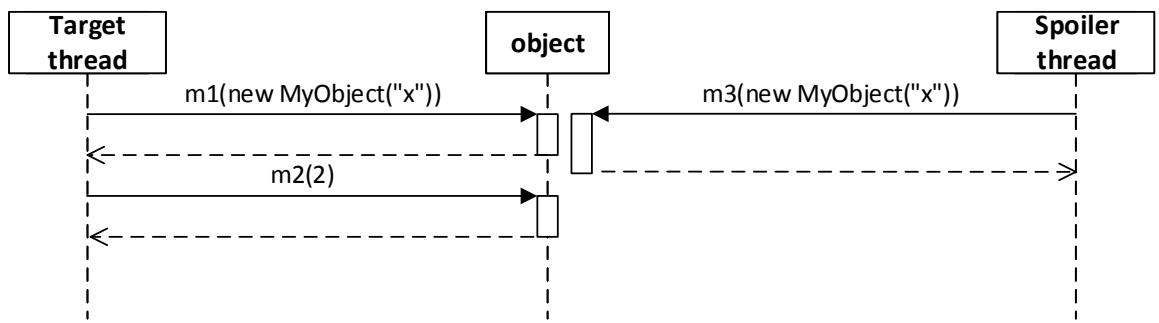
Obrázek B.33: Diagram testu 33.

Test 46

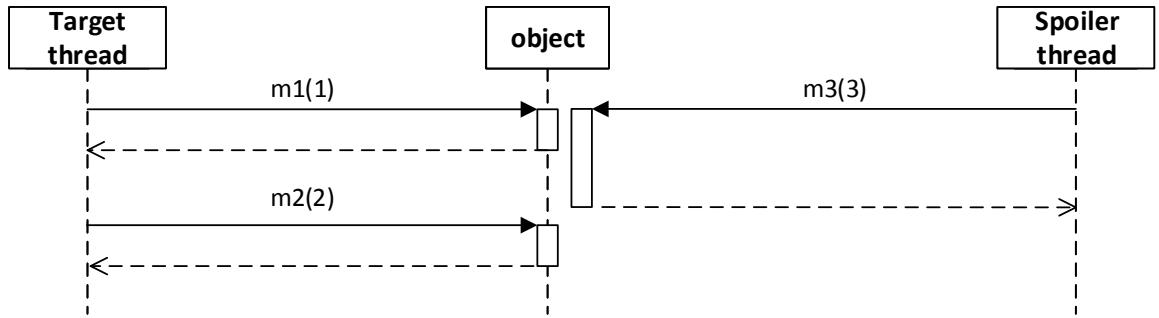
- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Dokončená instance targetu, ve vláknu *Spoiler thread*, je tvořena až druhými metodami *m1* a *m2*, protože by jinak nesouhlasí parametry s ohledem na definici kontraktu. Totéž platí u metody *m3* ve vláknu targetu *Target thread*.
- Kontrakt: $X:m1(_) \quad m2(A, X) \leftarrow \{ m3(_, X) \mid m1(A) \quad X:m4(A) \}$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



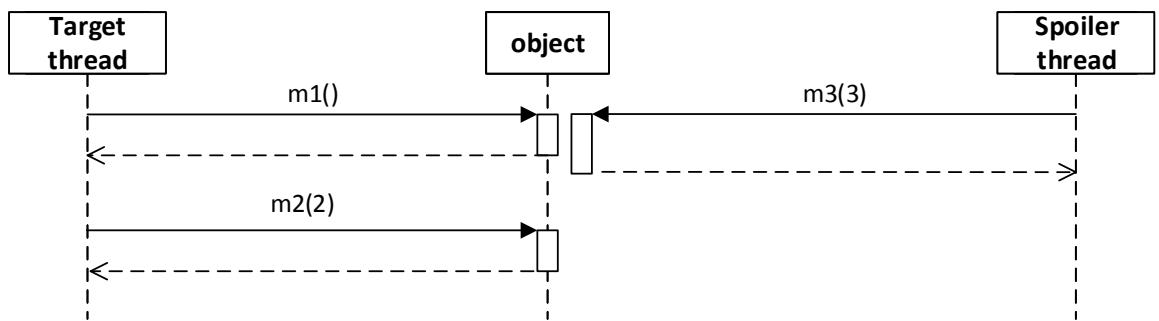
Obrázek B.34: Diagram testu 34.



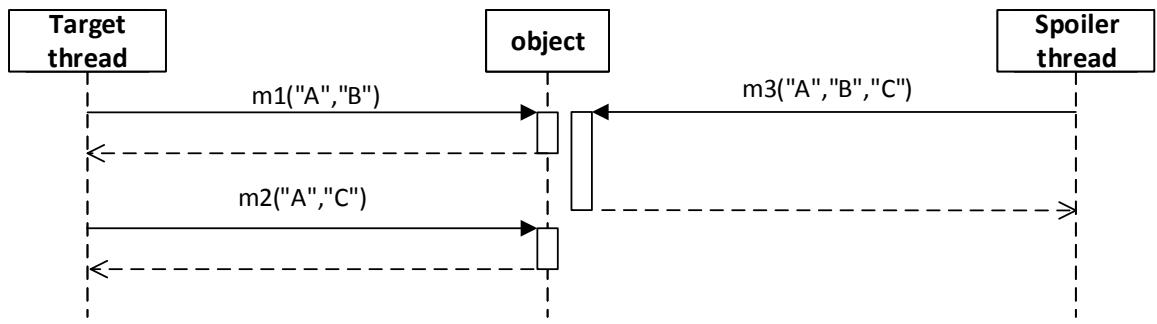
Obrázek B.35: Diagram testu 35.



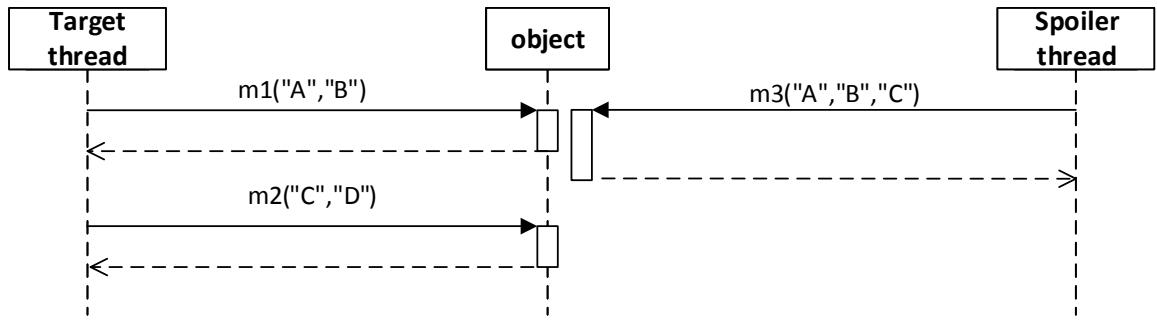
Obrázek B.36: Diagram testu 36.



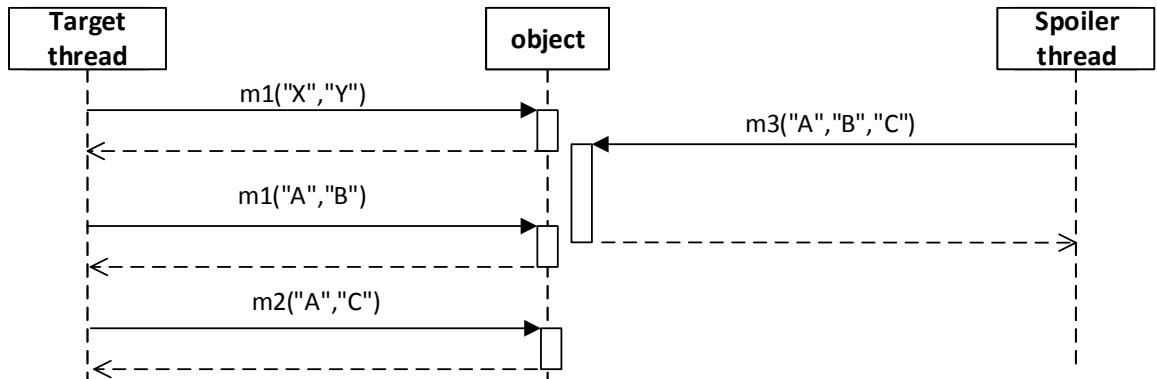
Obrázek B.37: Diagram testu 37.



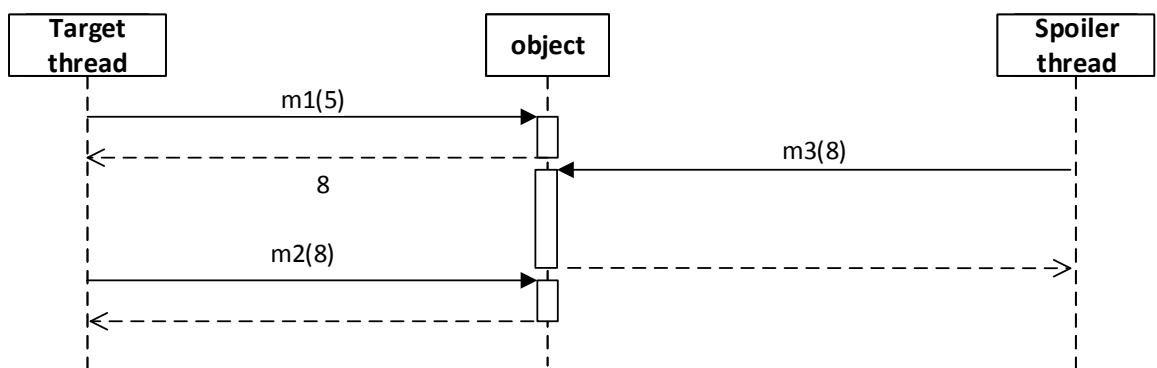
Obrázek B.38: Diagram testu 38.



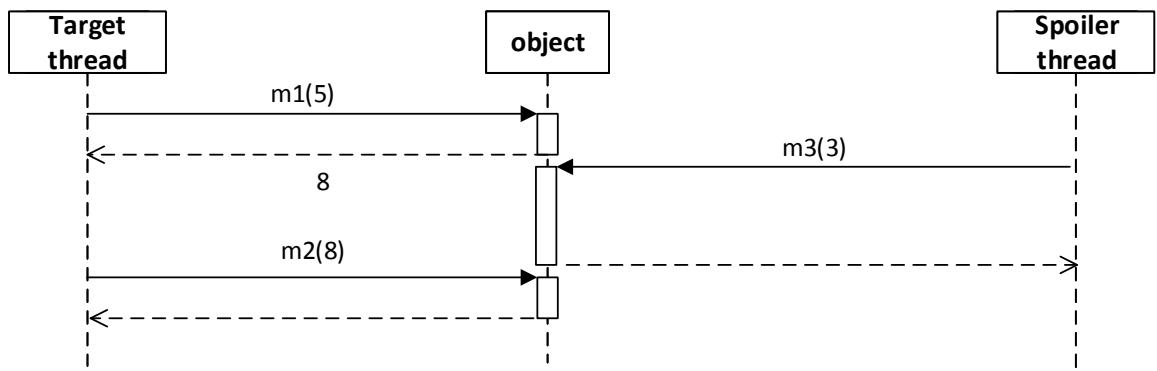
Obrázek B.39: Diagram testu 39.



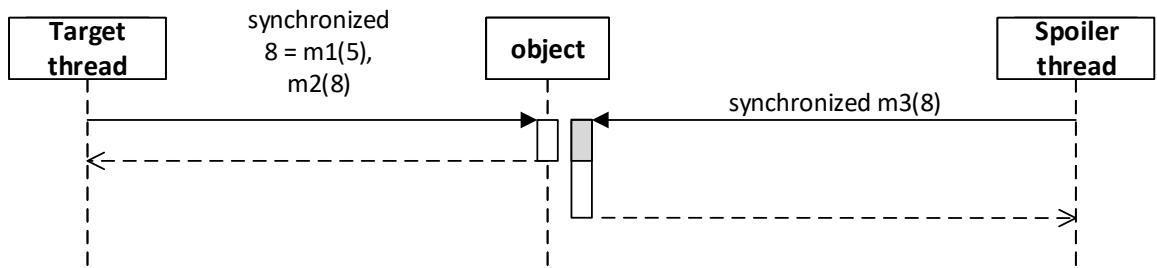
Obrázek B.40: Diagram testu 40.



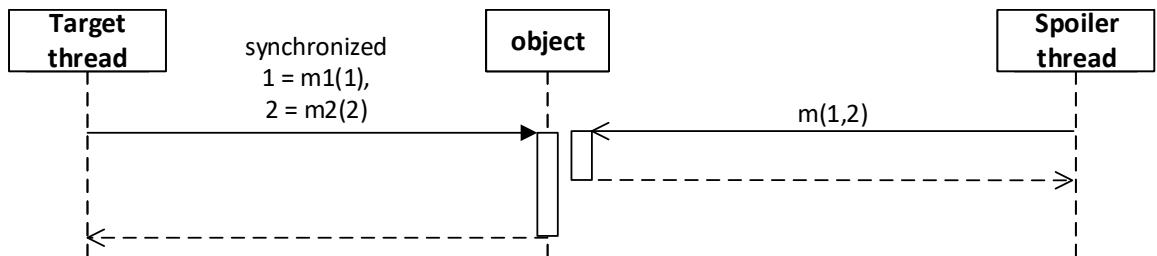
Obrázek B.41: Diagram testu 41.



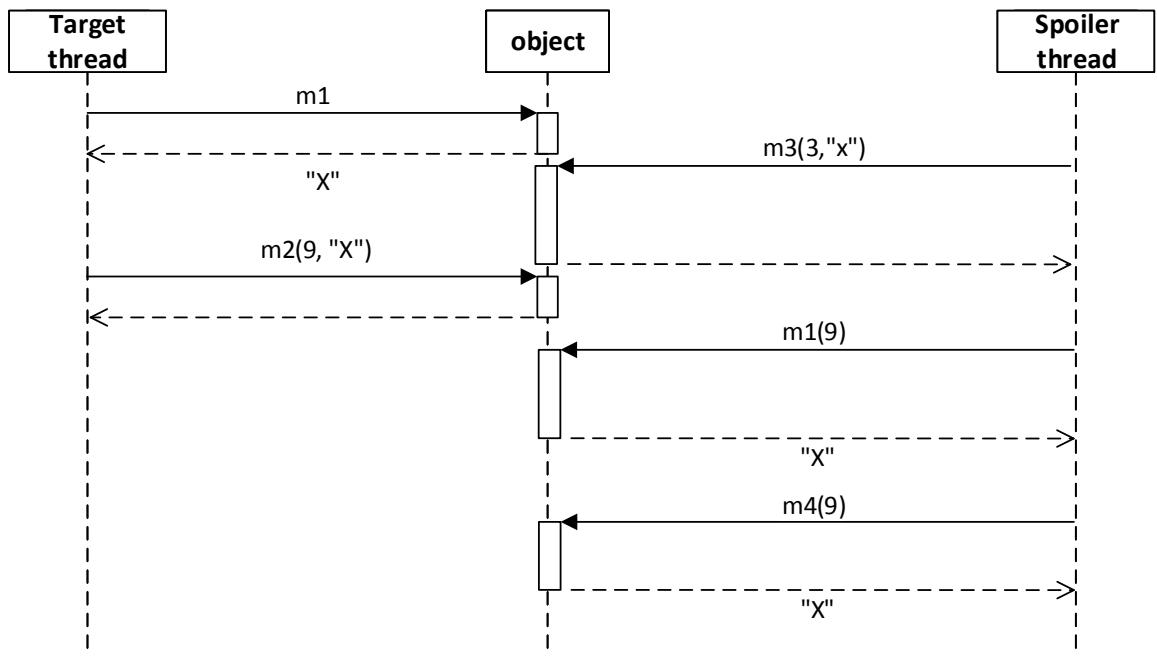
Obrázek B.42: Diagram testu 42.



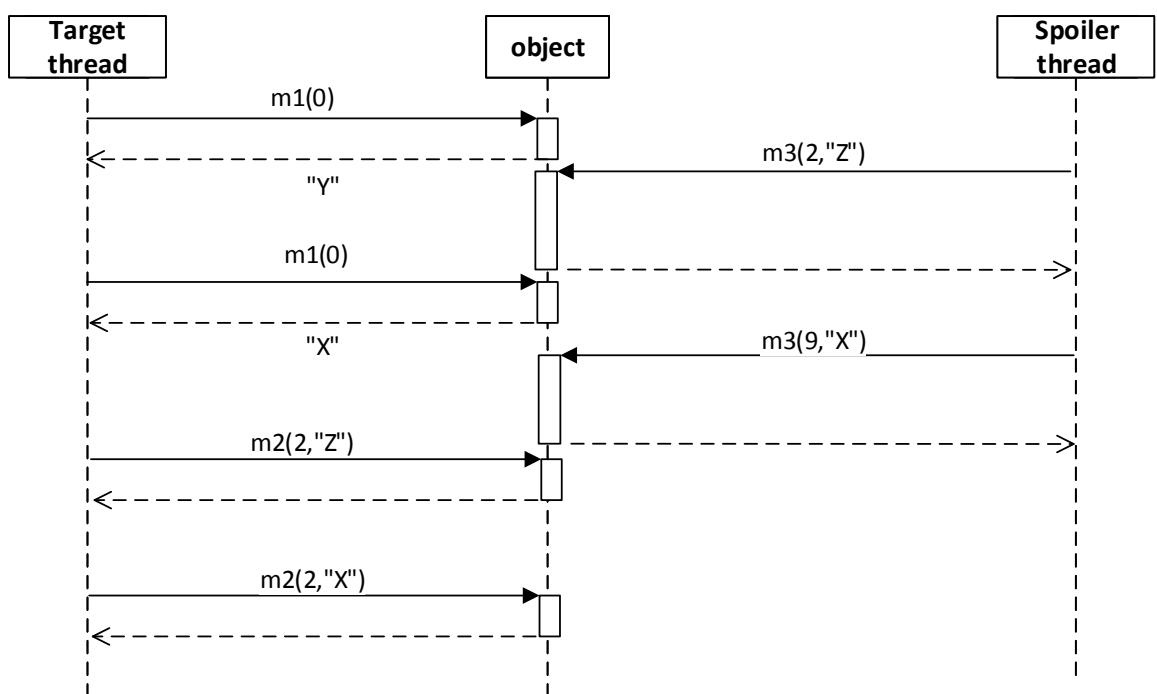
Obrázek B.43: Diagram testu 43.



Obrázek B.44: Diagram testu 44.



Obrázek B.45: Diagram testu 45.



Obrázek B.46: Diagram testu 46.